University of Utah
UNDERGRADUATE RESEARCH JOURNAL

ADDRESSING NOVICE CODING PATTERNS: EVALUATING AND IMPROVING A TOOL FOR CODE ANALYSIS AND FEEDBACK

Jacqulyn MacHardy Anderson (Dr. Eliane Stampfer Wiese)

School of Computing

ABSTRACT

Successful software needs maintenance over long periods of time. The original code needs to solve the problem and be maintainable. Computer science instructors at universities try to teach students in introductory classes how to write code that meets these goals. Results are inconsistent though. Readability is an important aspect of maintainability, and writing readable code requires choosing and structuring statements and expressions in a way that is idiomatic. Instructors of advanced CS courses perceive that their students frequently don't possess or don't engage this skill of writing with good structure. Building and evaluating this skill is complicated. Could students benefit from a tool that can identify and flag poor code structure and provide hints or suggestions on how to improve it while they program? Can a batch-processing version of the same tool help instructors quickly see and address the gaps in their students' understanding? These questions were investigated through think-alouds with ten students as they coded in Eclipse using PMD, a code structure analysis tool and interviews with four professors who teach introductory computer science courses.

In the think-alouds, students did produce novice code structures, and the tool performed as intended in flagging these structures and providing feedback. All students were able to correctly interpret the tool's feedback to successfully revise at least one of their initial implementations to use expert code structure. However, there is room for improvement in the tool's feedback.

The instructor interview results showed that although instructors believe that these patterns are important for students to learn and use, assessing and giving feedback on this aspect of student code by hand does not scale well to large class sizes, so their current grading and feedback processes do not target these kinds of code structures. Instructors believe these tools have the potential to address some of these gaps and challenges. Although no instructors want to interface with the output of the batch-processing tool the way it is currently presented, all of them were interested in seeing it extended and provided input on how the tools could be extended to better meet their needs.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to gratefully acknowledge the support and funding I was awarded by the Undergraduate Research Opportunities Program to conduct this research.

I would like to express my deepest gratitude to Dr. Eliane Wiese for her truly stellar mentorship. I have grown so much, inspired by her example and supported at every turn by her freely given wisdom and energy. Thank you :)

I would like to thank all my professors in the School of Computing. I would like to especially thank Dr. Parker who recommended me to Dr. Wiese in the first place! I would also like to thank Jamie King for teaching me to care about code style.

I would like to humbly thank all members of the HCI Research Group and Dr. Kopta for their contributions, genuine enthusiasm, generously given time, and insightful feedback on the research.

Many thanks go to the interview and think-aloud participants and to the School of Computing staff who helped with the logisitics of the interviews and think-alouds.

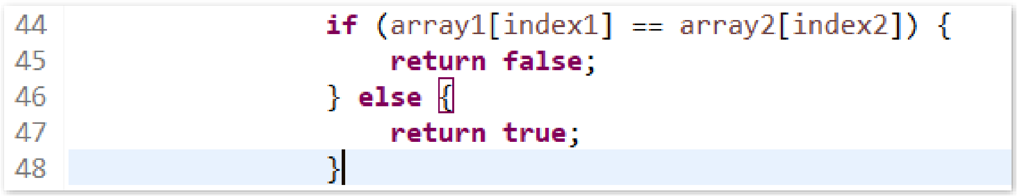
I would also like to thank my colleagues for the camaraderie and support throughout this journey. Special thanks go to my Senior Capstone Team!

I would like to thank my family who through their love and support and guidance smoothed out all the rough patches along the way.

Finally, I give my most heartfelt thanks to my husband. I am the lucky one.

INTRODUCTION

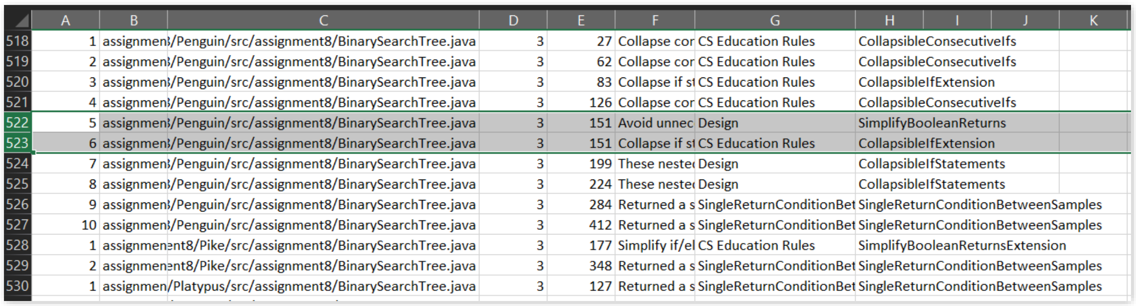
One important goal of writing a program is solving the programming task at hand. For novice programmers, sometimes this is the only goal [6]. On the other hand, maintaining and enhancing legacy code is far and away the dominant cost incurred during the life cycle of a successful software system [3]. Clearly, programmers must also pursue another goal: writing so that others can easily comprehend and modify the code in the future. Writing with good code structure in this sense is especially challenging for novices as their readability preferences tend to differ from experts’ preferences [12]. Research suggests that for certain coding patterns, students can comprehend both expert-structured and novice-structured programs equally well [12]. An example of a coding pattern the researchers examined is returning true or returning false instead of returning a condition directly (see Figure 1). Given that student comprehension was high for expert-structured and novice-structured code alike, Wiese et al. [12] concluded that it might be sufficient to detect novice patterns in student code and prompt the student to alter their implementation to improve its structure.



```
if (array1[index1] == array2[index2]) {
    return false;
} else {
    return true;
}
```

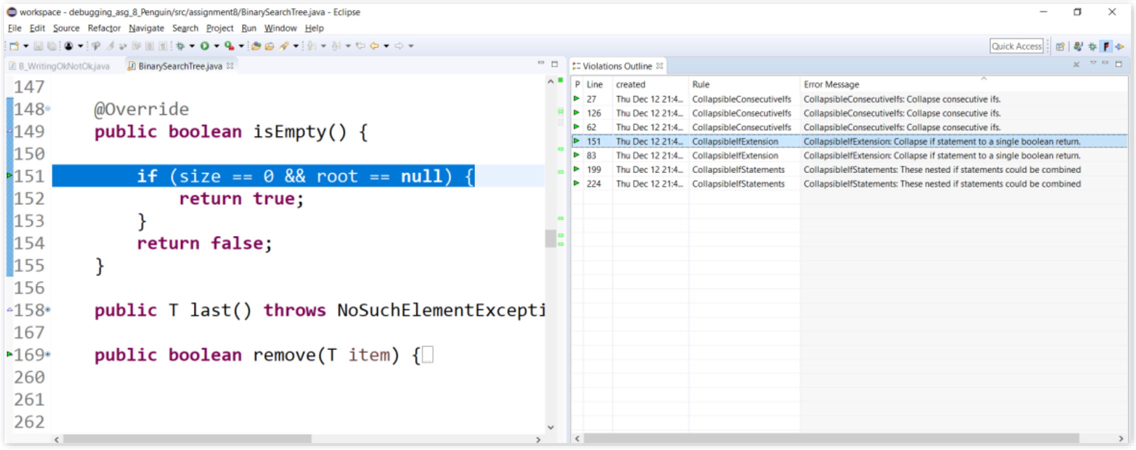
Figure 1 An example of a coding pattern. Boolean literals are returned where the condition itself could be returned instead.

One tool that could detect these kinds of patterns is PMD [5]. This tool is an Eclipse plug-in that statically analyzes Java code and flags lines of code that violate given code structure rules. This tool also has a batch processing version that is run from the command line and can process source code across multiple packages and directories. While other code structure guideline tools exist, they are not targeted to catch novice coding patterns and provide examples and suggestions in a way that is useful to a novice programmer [12]. This extended version of PMD is the tool that was used in the student think-alouds, and the batch-processing version used in the instructor interviews. See Figure 2 for an example of the output of the batch-processing version of PMD, and Figure 3 for an example of PMD's output in Eclipse that a student would see while programming.



	A	B	C	D	E	F	G	H	I	J	K
518	1	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	27	Collapse cor	CS Education Rules	CollapsibleConsecutiveIfs			
519	2	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	62	Collapse cor	CS Education Rules	CollapsibleConsecutiveIfs			
520	3	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	83	Collapse if st	CS Education Rules	CollapsibleIfExtension			
521	4	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	126	Collapse cor	CS Education Rules	CollapsibleConsecutiveIfs			
522	5	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	151	Avoid unnec	Design	SimplifyBooleanReturns			
523	6	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	151	Collapse if st	CS Education Rules	CollapsibleIfExtension			
524	7	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	199	These nester	Design	CollapsibleIfStatements			
525	8	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	224	These nester	Design	CollapsibleIfStatements			
526	9	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	284	Returned a s	SingleReturnConditionBet	SingleReturnConditionBetweenSamples			
527	10	assignmen	/Penguin/src/assignment8/BinarySearchTree.java	3	412	Returned a s	SingleReturnConditionBet	SingleReturnConditionBetweenSamples			
528	1	assignmen	ent8/Pike/src/assignment8/BinarySearchTree.java	3	177	Simplify if/el	CS Education Rules	SimplifyBooleanReturnsExtension			
529	2	assignmen	ent8/Pike/src/assignment8/BinarySearchTree.java	3	348	Returned a s	SingleReturnConditionBet	SingleReturnConditionBetweenSamples			
530	1	assignmen	/Platypus/src/assignment8/BinarySearchTree.java	3	127	Returned a s	SingleReturnConditionBet	SingleReturnConditionBetweenSamples			

Figure 2 Every issue that PMD flags in the input files becomes a row in the output table. As an example, the rows highlighted in this table correspond to the flagged code structure shown in the source code in Figure 3. This example also highlights a known problem where one pattern is flagged twice because two PMD rules overlap. The CS Education Rules category represents an extension of PMD [10].


workspace - debugging_asg_8_Penguin/src/assignment8/BinarySearchTree.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
B_WritingOkNotOk.java
BinarySearchTree.java
Violations Outline

```
147
148     @Override
149     public boolean isEmpty() {
150
151         if (size == 0 && root == null) {
152             return true;
153         }
154         return false;
155     }
156
158     public T last() throws NoSuchElementExcepti
167
169     public boolean remove(T item) {
260
261
262
```

P	Line	created	Rule	Error Message
▶	27	Thu Dec 12 21:4...	CollapsibleConsecutiveIfs	CollapsibleConsecutiveIfs: Collapse consecutive ifs.
▶	126	Thu Dec 12 21:4...	CollapsibleConsecutiveIfs	CollapsibleConsecutiveIfs: Collapse consecutive ifs.
▶	62	Thu Dec 12 21:4...	CollapsibleConsecutiveIfs	CollapsibleConsecutiveIfs: Collapse consecutive ifs.
▶	151	Thu Dec 12 21:4...	CollapsibleIfExtension	CollapsibleIfExtension: Collapse if statement to a single boolean return.
▶	83	Thu Dec 12 21:4...	CollapsibleIfExtension	CollapsibleIfExtension: Collapse if statement to a single boolean return.
▶	199	Thu Dec 12 21:4...	CollapsibleIfStatements	CollapsibleIfStatements: These nested if statements could be combined
▶	224	Thu Dec 12 21:4...	CollapsibleIfStatements	CollapsibleIfStatements: These nested if statements could be combined

Figure 3 PMD 's feedback appears in the source code editor and in a separate "Violations Outline" view in Eclipse.

BACKGROUND AND RELATED WORK

Research suggests that novices tend to find novice-structured code more readable, even though they are capable of comprehending expert-structured code and novice-structured code equally well [12]. Why do students prefer to read and write in a novice-structured way when they are capable of comprehending expert-structured code? One hypothesis that may explain this mismatch is that if students do not completely understand what their program is supposed to achieve, or perhaps they aren't familiar enough with the language to know what data structures and functions exist that could help them solve the problem in a simpler way, then they will write code that is unidiomatic to an expert [11]. Prompting students with actionable structure feedback while they are coding could help them more effectively learn these idioms, thereby addressing at least one of the issues that leads to novice-structured code.

Wiese et al. [13] did some initial exploration of the idea that simply prompting students with suggestions or examples might be sufficient to lead them to improve the structure of a piece of code. To explore this, they prompted students to edit existing code to improve its structure. They found that despite being able to recognize expert structure for a given pattern, for example, returning a Boolean condition directly instead of checking the condition and then returning Boolean literals, many students were not able to correctly revise novice-structured code. It's possible that editing someone else's code (as opposed to revising one's own code) presented a barrier.

The form and level of feedback a student needs from an IDE is not the same as what a professional programmer needs. For example, when an error message is presented from the perspective of the IDE rather than the perspective of a student (e.g. “Error: expected x but found y”), the student may face a barrier to understanding and resolving it as they may not have an accurate mental model of the process by which their code was parsed to generate that message [8]. While other structure feedback tools besides PMD (2002) do exist, such as Style Avatar [7], FrenchPress [1], Submitty [9], and StyleCop [2], they do not detect all of the common novice patterns identified by Wiese et al. [12]. Wiese et al. [12] note that many current tools either focus on typographical issues only, or they target complex, project-scale concerns that are relevant only to experienced programmers. Another tool, AutoStyle [14] is targeted to the novice programmer, but in providing feedback it does not account for novice readability preferences.

RESEARCH QUESTIONS

A main research question addressed in this thesis is: could students benefit from a tool that can identify and flag poor code structure and provide hints or suggestions on how to improve it while they program? This question was investigated through think-alouds with ten students as they coded in Eclipse using PMD, a code structure analysis tool. One goal of the think-alouds was to see how students respond to and think about PMD-style feedback. PMD's strong customizability and extensibility make it well-suited for supporting students as it can be tailored to their specialized needs.

The other main research question addressed is: can a batch-processing version of PMD help instructors quickly see and address the gaps in their students' understanding? This question was addressed through interviews with four professors who teach introductory computer science courses.

THINK-ALOUD METHODS

Participants

Participants were recruited from the pool of students in CS 2420 and 3500 who had previously participated in a code structure study and had agreed to be contacted via email about a follow-up study. Think-aloud participants were compensated $15 for participating for one hour.

Methods

Think-aloud participants were provided the Eclipse IDE on a Mac computer, and Eclipse already had the PMD code checker plug-in installed. The tasks were prepared ahead of time and presented as a collection of method signatures, Javadoc comments describing what the method was supposed to do, and JUnit tests. The JUnit tests were not comprehensive, meaning that a student could pass all the tests without having completely fulfilled the requirements laid out in the Javadoc comments. To begin the think aloud, students were told the task steps, which were coding, testing correctness, fixing bugs, checking style, and revising style. Students were also shown how to run the provided unit tests. Students were then provided a Java class with a method stub and instructed to implement the method according to the provided Javadoc comment. They were asked to think out loud while implementing the method as described in the Javadoc, and they were allowed to ask clarifying questions about the Javadoc as needed.

When they seemed to be finished coding, if they did not immediately run the JUnit tests themselves, they were prompted to run the tests. If the tests failed, they had to interpret the failure and resolve it. They were offered debugging assistance only if they got stuck. For example, a student who was out of ideas on how to diagnose the problem might be instructed to look at the relevant unit test or to put a breakpoint on the first line of the method and run the code in Debug mode.

When their implementation of the first method passed the unit tests, they were introduced to PMD and shown how to run it. They were told that PMD was a “style checker,” and each step of running it was explained out loud as it was run on their implementation: “Right-click on the Java file, go to PMD, and click on ‘Check Code.’” If PMD had feedback, the mouse was pointed at the in-line feedback, and the student was told that this was the feedback from PMD. For all subsequent methods that the student implemented, if they did not immediately run PMD on their code, they were prompted to run it. If PMD had no feedback but their implementation was novice-styled, they were prompted to revise their code style. If they stopped thinking out loud, they were prompted to voice their thoughts or opinions on what they were reading or doing, or they were asked to continue thinking out loud.

THINK-ALOUD RESULTS

This thesis includes analysis of two programming tasks where students were asked to implement code from scratch, test their implementation's correctness using provided unit tests, fix bugs caught by the tests, check their code's style using PMD, and finally revising their code's style based on PMD's feedback. There were additional programming tasks included in the think-aloud that are not included in the analysis in this thesis. Six students were asked to implement `isStrictlyPositive`, a method that returns true when the inputted int is positive (Appendix: Think-Aloud Tasks). This task targets the Simplify Boolean Returns pattern (Appendix: Coding Patterns). See Table 1 for a summary of PMD's expected and actual performance on each variation of the Simplify Boolean Returns pattern.

Table 1 PMD performed as expected in detecting one version of the Simplify Boolean Returns coding pattern. Not all versions targeted were generated by participants, and one version generated by a participant was completely unanticipated.

Simplify Boolean Returns Versions (Appendix: Coding Patterns)	PMD Expected Performance	PMD Actual Performance
Version 1	PMD expected to detect.	Three participants produced. PMD did detect.
Version 2	PMD expected to detect.	No participants produced.
Version 3	PMD expected to detect.	No participants produced.
Version 4	Version not anticipated prior to think-aloud. PMD not programmed to detect.	One participant produced. PMD did not detect.

Of the six students who completed `isStrictlyPositive`, five were then asked to implement `okNotOk`. This is a method that returns "Ok" when the result of the first inputted int divided by the second inputted int is greater than or equal to seven and the product of the two inputted int's is greater than or equal to one hundred twenty eight, but returns "Not Ok" otherwise. This task targets the Collapsible If Statements pattern (Appendix: Think-Aloud Tasks). See Table 2 for a summary of PMD's expected and actual performance on each variation of the Collapsible If Statements pattern.

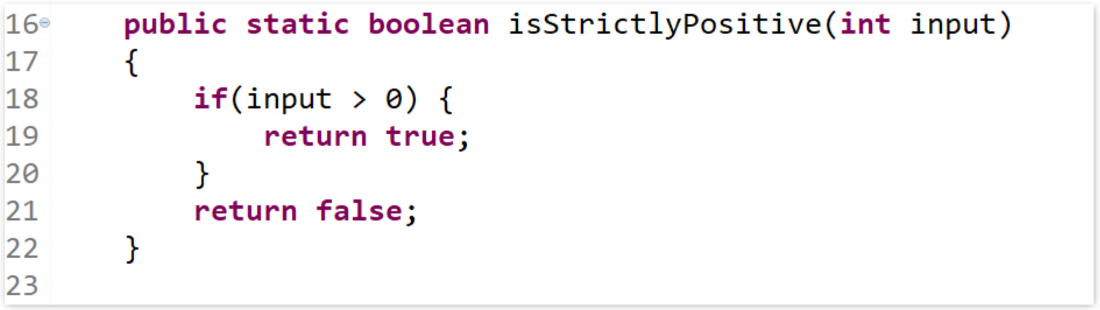
Tool Effectiveness

Students Produced Patterns the Tool Should Flag and the Tool Flagged Them Correctly

PMD flagged the code for Participants 1, 3, and 4 in `isStrictlyPositive` task; their code looked like Figure 4. The feedback it provided was the same for each of them: "Avoid unnecessary if..then..else statements when returning Booleans." For the `okNotOk` task, no participants produced the nested `if`-statement version of the Collapsible If Statement pattern that PMD is programmed to detect.

Table 2 No students generated the targeted versions of the Collapsible If Statement pattern. One version generated by three participants was completely unanticipated.

Collapsible If Statements Versions (Appendix: Coding Patterns)	PMD Expected Performance	PMD Actual Performance
Version 1	PMD expected to detect.	No participants produced.
Version 2	PMD expected to detect.	No participants produced.
Version 3	Version not anticipated prior to think-aloud. PMD not programmed to detect.	Three participants produced. PMD did not detect.



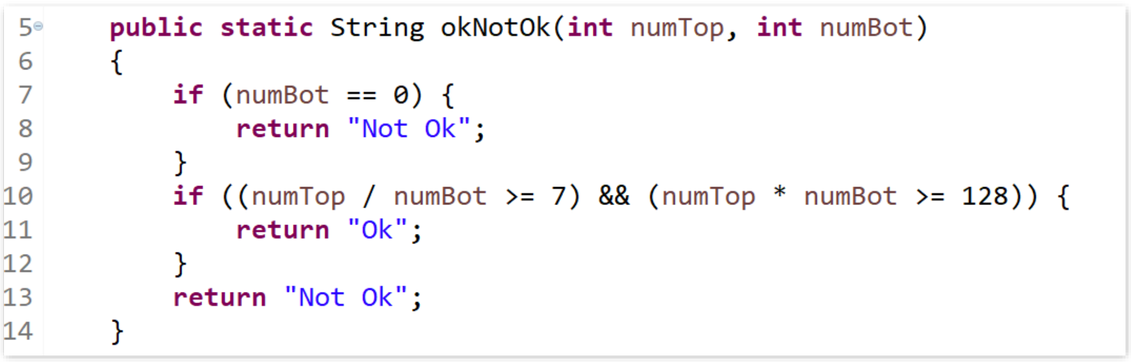
```
    public static boolean isStrictlyPositive(int input)
    {
        if(input > 0) {
            return true;
        }
        return false;
    }

```

Figure 4 A novice code structure commonly produced during the isStrictlyPositive task. Falls under the Simplify Boolean Returns novice pattern and was flagged by PMD.

Participants 1, 2, and 3 wrote code that looked like Figure 5 once they made their code pass the provided unit tests. This code does fall under the Collapsible If Statements pattern because the `if`-statements on lines 7 and 10 could be collapsed into a single `if`-statement, as in Figure 6. However, PMD is not yet programmed to detect this pattern the way it appeared in the students' code, where the `if`-statements are consecutive. Participants 5 and 6 initially wrote code that used good structure and was not flagged by PMD, but failed a provided unit test (see Figure 7). Participant 5 revised their code to use expert structure (similar to Figure 6). Their conditional had a bug, but it passed the provided unit tests. Participant 6 revised their code to look like Figure 8, which does not violate the Collapsible If Statement pattern and was not flagged by PMD, but still might not be considered idiomatic by experts.

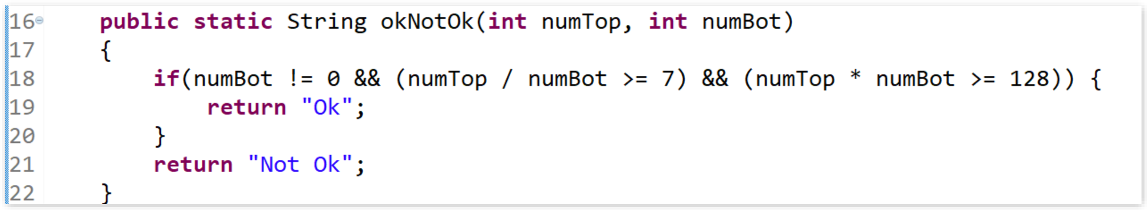
Students' Interpretations and Implementations of the Tool's Feedback

Participants 1, 3, and 4 each received the same feedback for their initial `isStrictlyPositive` implementation (see Figure 4), but each of them had different interpretations of the feedback, which suggests that the current form and phrasing of the



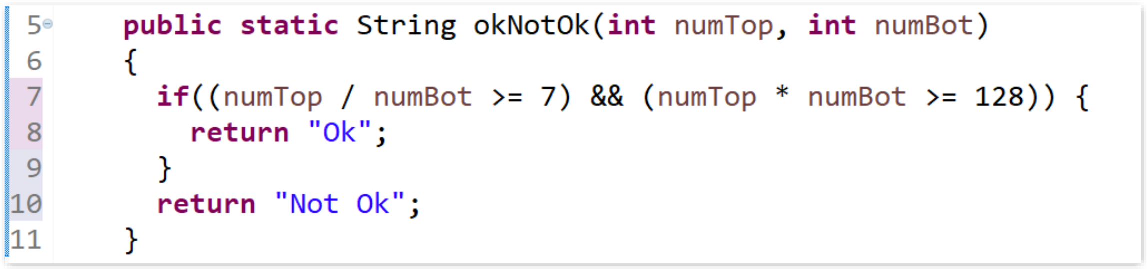
```java
public static String okNotOk(int numTop, int numBot)
{
    if (numBot == 0) {
        return "Not Ok";
    }
    if ((numTop / numBot >= 7) && (numTop * numBot >= 128)) {
        return "Ok";
    }
    return "Not Ok";
}
```

Figure 5 An interesting novice code structure that PMD was not programmed to detect. Commonly produced during the okNotOk task. Falls under the Collapsible If Statements novice pattern and was pointed out to the student by the author.



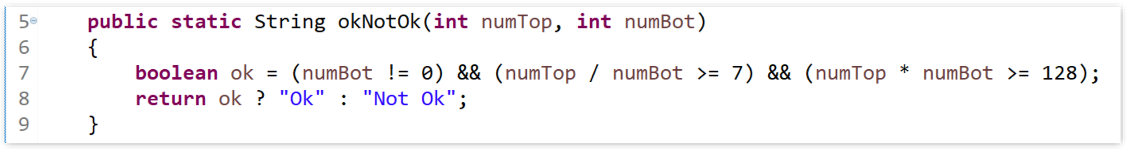
```java
public static String okNotOk(int numTop, int numBot)
{
    if(numBot != 0 && (numTop / numBot >= 7) && (numTop * numBot >= 128)) {
        return "Ok";
    }
    return "Not Ok";
}
```

Figure 6 Expert-structured code with logic that matches the okNotOk Javadoc.



```java
public static String okNotOk(int numTop, int numBot)
{
  if((numTop / numBot >= 7) && (numTop * numBot >= 128)) {
    return "Ok";
  }
  return "Not Ok";
}
```

Figure 7 A common implementation of okNotOk. Throws a DivideByZeroException when numBot is 0.



```java
public static String okNotOk(int numTop, int numBot)
{
    boolean ok = (numBot != 0) && (numTop / numBot >= 7) && (numTop * numBot >= 128);
    return ok ? "Ok" : "Not Ok";
}
```

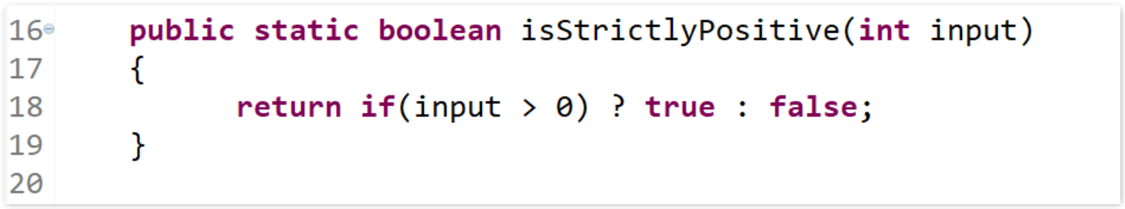
Figure 8 An interesting novice code structure that was produced by one participant during the okNotOk task. Does not fall under any of the current patterns.

feedback might be insufficient for a student who is unfamiliar with the code structure patterns used in this study. Participant 1 interpreted it correctly and confidently, Participant 3 interpreted correctly but was not confident, and Participant 4 interpreted it incorrectly and confidently.

Participant 1 read the PMD feedback out loud, and then stopped talking. They were asked whether they could incorporate it into a revision. They said, “Yeah, I know what you want,” indicating that PMD had successfully alerted them to the structure issue and that they understood how to revise it based on PMD’s feedback. They revised their code successfully, saying, “I can just do this in one line…”

Participant 3 read PMD’s feedback silently. Then they said, “Hmm, yeah, ok, so I could probably just do something… else to make it shorter? I don't know,” suggesting that they were unsure what the problem was but they thought the code could be shorter if they somehow removed the “unnecessary if..then..else statements.” While they revised their code, they said, “That would do… the same thing, I guess,” indicating that they were unsure about the equivalency of their new solution and their original solution.

Participant 4 read the feedback out loud and without prompting or pausing, they continued, “Ok, so then... they say uh, how do you use the ternary operator?” suggesting that they thought the problem PMD was highlighting was the use of if-else statements when the actual problem was the returning of Boolean literals where a Boolean expression would suffice. They revised their implementation to use the ternary operator (Figure 9) and then ran PMD unprompted. The feedback disappeared which was actually a false negative. PMD is not set up to detect ternary operators, but their code still exhibited the “unnecessary if.. then.. else” statements, just in compacted form.



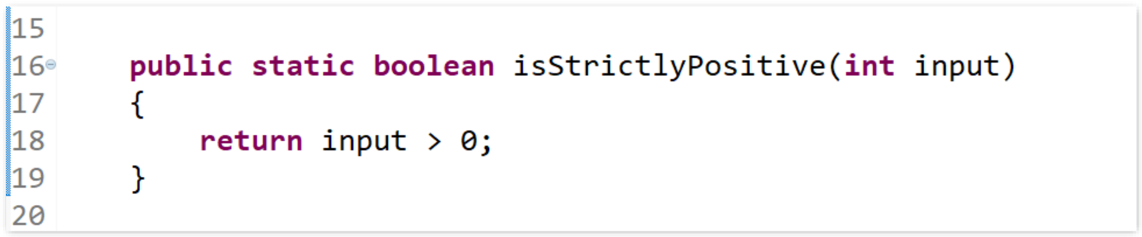
```
public static boolean isStrictlyPositive(int input)
{
        return if(input > 0) ? true : false;
}
```

Figure 9 An interesting novice code structure that PMD was not programmed to detect. Produced by one participant during the isStrictlyPositive task. Falls under the Simplify Boolean Returns pattern and was pointed out to the student by the author.

Usability

On the first task, `isStrictlyPositive`, participants 2, 5, and 6 received no feedback from PMD on their initial implementation. Their initial implementation (see Figure 4) exhibited expert structure. When they encountered this lack of feedback, they did not react at first, indicating that they did not realize that the tool had finished running. It was explained to them that the tool had finished running, and the lack of feedback meant that PMD had not flagged anything in their code.

On the second task, `okNotOk`, Participants 2 (Figure 5) and 6 (Figure 8) both ran the style checker unprompted after running the JUnit tests. When PMD showed no feedback, Participant 2 asked, “Is it done?” suggesting that even though the meaning behind the lack of feedback had been explained to them during the first task, it was still confusing to the student to see no indication that PMD had finished running. When PMD showed no feedback for Participant 6, they ran it again. When PMD still had no feedback, they said, “Well either it doesn't have any complaints or it's broken,” indicating that the lack of feedback was completely ambiguous to them, just as it was to Participant 2. Participant 3 received feedback from PMD in the `isStrictlyPositive` task, and even after they modified their code to correctly addressed the style feedback (Figure 10),



```
    public static boolean isStrictlyPositive(int input)
    {
        return input > 0;
    }
```

Figure 10 Expert-structured code with logic that matches the isStrictlyPositive Javadoc.

PMD's feedback did not change when they ran it again. They did not react, suggesting that they were waiting for an indication that PMD had finished running. PMD had presented the same feedback because the participant had not saved their changes. This was explained to them and they saved it and ran PMD again, making the feedback disappear. The student said, "Ok, now it's ok, I think," again suggesting that they were not completely sure what the PMD feedback or lack of feedback meant.

<u>Relationship Between Problem-Solving Approach and the Pattern Produced</u>

Participants 3 and 4 went back and forth between reading the Javadoc out loud and typing a small piece of their solution. For example, Participant 3 read out-loud, "This method takes in a Boolean and returns an int." The student then typed return false into the method body, saying "So, ok, I'm just going to return false so I remember," They continued reading, "'Returns true if the following condition is met, the input is positive.' So, I'll just say if input is greater than ... so if it is positive, so zero is not technically positive, so if input is greater than zero return true." This direct, one-to-one translation of the Javadoc into logic indicates that these participants did not have a separate planning step where they considered multiple code structures and weighed their decision before finally choosing a particular structure. Of the three participants who initially produced a correct solution with novice structure, two coded with this one-to-one approach. Only one of the three (Participant 1) finished reading it first, saying, “Well the test is if the input is positive, so ‘if input greater than 0, return true’…” Then, examining the code and adding whitespace, they finished, “So the only other situation is to return false.” Even though they finished reading the Javadoc before they started typing, their words suggest that they were thinking of the problem as a direct translation from the Javadoc comment.

In contrast to the one-to-one approach, Participant 2 first read the Javadoc out loud from start to finish, and then started to type `if(input > 0)`. As they were typing, they paused and said, “Well, actually, I’m rethinking this,” indicating that, even though their code was correct, they had noticed something about it that could be improved, and they understood how to make the improvement. They deleted what they had and typed `return input > 0`. This indicates that the student was aware of their code structure choices while they were drafting their solution; they spontaneously recognized and revised a structure issue without any external prompting. Participants 5 and 6 also produced correct, expert-structured code without feedback or prompting. They finished reading the Javadoc and then summarized it in their own words, indicating they had synthesized the meaning behind the requirements before typing their solution. For example, Participant 6 said, “Ok, so basically, literally all I'm going to do is return input strictly greater than zero.” They did hesitate when returning the Boolean expression. Participant 6 was unsure if parentheses were required, and Participant 5 was unsure whether the Boolean expression could be returned at all.

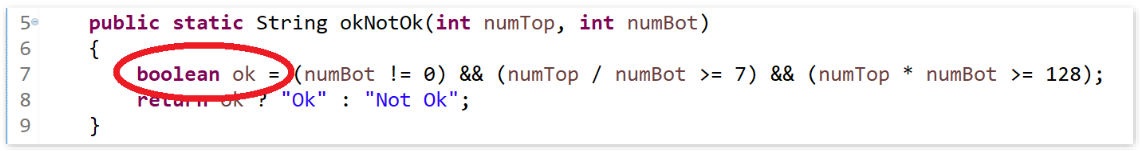
For the `okNotOk` task, Participants 3 and 4 coded with the same approach as before, reading a little bit of the Javadoc, typing a literal translation of what they’d read, and repeating this cycle until they finished reading and implementing the Javadoc. Participant 1 stated that they were, “going with [their] first instincts,” suggesting that even though they finished reading the Javadoc before they started typing, they were not engaging in planning or revising steps. Participant 2 read the Javadoc out loud, then started coding, saying, “Ok, um, so I'm gonna calculate the result first, int result... well, maybe I don't need to do that. I am overthinking this,” suggesting that they were going through a planning and revising stage in their head. Only Participant 2 passed all the unit tests in their first attempt (Figure 5).

The other four participants’ initial implementations (Figure 7) failed the test that exercised their code’s handling of the case where the argument for the parameter `numBot` was zero. This case led to a zero in the denominator of a division operation for the participants who failed this test, causing a `DivideByZeroException` to be thrown, which is not aligned with the instructions in the Javadoc comment. Participant 6 read the test results and began to think out loud, planning two different potential structures for their solution, which suggests that style is important to them and that planning is an integral part of their process. Figure 8 shows Participant 6’s final implementation. Participants 1 and 3 both revised their code to look like Figure 5, and commented on their code structure as they were revising, saying, “there is probably a more efficient way to do this,” and “I don't like that this is in there twice, but... off the top of my head I can't think of a better way to do this,” suggesting that they both recognized that there were style issues but did not feel confident about addressing them before validating their code’s correctness.

Attitudes

About Style

After being prompted, Participant 2 thought out loud about the style of their `okNotOk` implementation (Figure 5) which used two consecutive if-statements that could have been collapsed into a single statement, a version of the Collapsible If Statement pattern, concluding, “I tend to think of error checking as its own kind of if statement and then anything that's method specific is its own if statement itself 'cause any error checking you're gonna wanna automatically return or throw an exception, so I tend to think of them in two separate realms, I guess.” This is important because it suggests that it's possible for students to develop their own structural idioms and to have a sophisticated intent behind their style choices that needs to be respected and not arbitrarily “corrected.” Along the same lines, when Participant 6 was asked how they chose one implementation over another while they were planning, they explained, “I prefer code that is more concise, and just adding this [assignment to a local variable], which is maybe 10 characters, as opposed to adding a new [statement]- the weight of a [statement] is more than the weight of characters within a [statement].” See Figure 11 for a highlighted example that calls out the assignment in the code that the student referenced. Once again, these two students’ responses together suggest that students may develop their own meaningful, structural idioms and guidelines, though the code structures they produce may look nothing alike and may not use expert structure.



```java
public static String okNotOk(int numTop, int numBot)
{
    boolean ok = (numBot != 0) && (numTop / numBot >= 7) && (numTop * numBot >= 128);
    return ok ? "Ok" : "Not Ok";
}
```

Figure 11 Participant 6 intentionally chose to assign the condition to a local variable rather add a new if-statement.

About the Tool

When being introduced to PMD, Participant 4 was told, “We have something here that checks code style,” to which they said, “Oh really? That's really cool!” indicating that they were enthusiastic about the concept. Participant 3 ran PMD again after their structure revisions, unprompted, saying, “Let's see if PMD still likes me.” PMD had no feedback, and observing this, the student said, “Seems like it does,” echoing the way they had talked about PMD earlier in the task when they said, “If it yells at me that’s ok.” The way they anthropomorphized PMD suggests they developed a certain rapport with it, even though they had only used it briefly. When Participant 4 revised their `isStrictlyPositive` implementation to use the ternary operator, they said, “well, ok. That should hopefully solve the feedback,” suggesting that they were thinking of the PMD feedback as a single, external problem that needed to be solved.

Additional Observations

After resolving PMD’s feedback for `isStrictlyPositive`, Participant 3 asked if the code runs slightly faster without the `if`-statement, showing an interest in the performance implications of code structure choices. After revising, Participants 1, 3, and 4 all made very similar statements, declaring that they were going to let the unit tests determine the correctness of their code for them. This marked an interesting shift in the writing process where their perception of the task seemed to change from “implement this method based on this Javadoc” to “pass these unit tests.” For example, one student said, “... I'm going to run the tests to see if it works.” Another took several tries to correct the error, and along the way, they would make a small change and then run the tests again without pausing to evaluate the correctness or style of the code themselves. This pattern is interesting because it might be indicative of student modes of interaction with the PMD style tool if it becomes ubiquitous in the same way unit tests have.

Participant 4 resolved PMD’s feedback for `isStrictlyPositive` by using a ternary operator (see Figure 9). Seeing the feedback disappear, they said, “So that made it work? I mean, it's cleaner code, so that makes sense,” suggesting that although they had correctly interpreted the disappearance of the feedback, they had also over-interpreted the false negative to reinforce their concept of cleaner code. They were then prompted to see if they could make the code even cleaner, to which they responded, “Oh! Yeah, I'm overcomplicating, aren't I?” Without any example or explanation, just a suggestion to try, they utilized the expert pattern and returned the Boolean expression directly, indicating that they had enough experience to identify and revise a structure issue using their own judgment when given a simple, generic prompt to improve their code.

Participant 1 was reminded that they had stated that they didn't like that they were returning “Not Ok” in two different places in the code for the `okNotOk` task. In response, the student revised their code, pointing out that they were making use of short-circuiting. Participant 3 was prompted similarly, and in response they started to brainstorm out loud, saying that they thought they could cover all the “Not Ok” cases in one `if`-statement and then return “Ok” in the `else` case.

Both participants revised their code to look like Figure 6 without any additional prompting. In both cases, this revision required them to flip the condition of their previous fix (so, `numBot != 0` instead of `numBot == 0`), suggesting that they were thinking about the correctness implications of their style revision, not carelessly moving chunks of code around.

THINK-ALOUD DISCUSSION

Think-aloud results that were enriched when contextualized by the instructor interviews are addressed later in the Combined Discussion. Think-aloud results which were interesting and useful independent of the instructor interviews are analyzed here.

Instruction Recommendations

More of the students who read the entire Javadoc before coding produced correct, expert-structured code than did students who solved the tasks with a one-to-one approach (e.g. reading a line of the Javadoc and then writing a line of code). This suggests that instructors should emphasize the benefits and importance of coding in a systematic way with discrete steps, as part of teaching students to use good style. Some examples of systematic approaches that students used successfully during think-alouds were: saying pseudocode out loud before writing actual code, reading all available information and summarizing it, or explicitly formulating a starting point and an end point for the problem before defining the path of the solution.

Some students already care about style and have well-reasoned, sophisticated intent behind style choices, even those choices that contradict the expert code structures. On the other hand, some students seemed to just care about making PMD happy. They saw style as a personal preference, not as a vehicle for intent, and PMD’s feedback did not necessarily reshape their personal preferences. They simply made the necessary changes to make the feedback go away. It is important that instruction on style not be prescriptive. Some leeway needs to be afforded for thoughtfully argued style choices, and some instruction time may need to be devoted to introducing students to the idea that there are tangible arguments for writing code a certain way. For example, code structure carries meaning for the reader, and well-chosen structures can prevent or expose bugs.

Tool Design Recommendations

One student encountered a usability error where PMD did not re-run because they had not saved their changes. An experienced programmer encountering this issue would likely be able to narrow down the reason for this behavior, but for a novice programmer who is using PMD for learning, this is problematic. It was not apparent to them that the tool had not run, and the student was confused that their changes did not cause any changes in PMD’s feedback. This could be improved either with an error message, alerting the student to the fact that the tool had not run because they had not saved, or by providing the student some kind of very obvious visual cue when PMD does run, separate from the feedback.

INTERVIEW PARTICIPANTS AND METHODS

Instructors who have taught CS 1410, 2420, or 3500 were recruited for interviews. During recruitment, instructors were asked to bring the following to the interview: rubrics for two programming assignments where some aspect of programming other than functionality/performance was part of the rubric (one assignment from the beginning of the semester and one from the end, from their most recent offering of CS 1410 or 2420 or 3500); examples of anonymized assignments with low vs. high scores to demonstrate how the rubric was applied. Reviewing and discussing the rubrics was the first phase of the interview. Instructors were asked a mix of prepared questions and spontaneous follow-up questions, with the goal of understanding the instructor's current process for grading and giving feedback on "style" and code structure in programming assignments, separately from their evaluation of the correctness of student code.

Following this, instructors were shown the novice coding patterns [12] (Appendix: Instructor Interview Coding Patterns Document). They were asked to identify which patterns were important to them as instructors and which patterns they observed in student code. Some patterns were presented and discussed without the concrete examples shown in the Appendix, but through discussion it was clear the instructors understood the code structures the pattern names referred to. Instructors then received a demo of the instructor-facing PMD command line tool which batch processes Java source code and outputs a file that contains details for every line of code that was flagged by PMD (see

Figure 2). The instructors were asked to interpret the output of the tool, and they were asked a mix of prepared questions and spontaneous follow-up questions with the goal of gauging how the instructor might incorporate the tool into their grading and feedback process, how they would want to see the tool extended, and what feedback they had about the tool and its output.

Time and interest permitting, instructors also received a demo of the student-facing PMD Eclipse plug-in, and they were asked for their feedback on the usefulness of the tool, their thoughts on how they would like to see it extended, and their thoughts on how they might incorporate this tool in their teaching and feedback process. At the conclusion of the interview, instructors were given a final opportunity to critique the instructor-facing tool.

INTERVIEW RESULTS

How Do Instructors Teach, Evaluate, and Give Feedback on Style?

The instructors' current processes for evaluating student code structure and style was the first focus of the interviews. Three of the four instructors interviewed provided at least two homework assignment rubrics that included at least one rubric item that focused on some aspect of the code other than correctness. These rubric items tended to be things like "Style & Design" or "Documentation & Style," and students would earn points in these categories for things like following instructions to replace certain comments with specific information, using helper methods, including Javadoc comments on every method, class, and instance variable, using whitespace and brackets consistently, and including in-line comments where the code was "unusual or complex" (citing a rubric from CS 2420). The current rubrics do not explicitly address code structure, and all instructors noted that any specific feedback provided to the students regarding their code style and design was dependent on the judgement and time constraints of the TA's.

Although these code structures do not get addressed in assignment rubrics, instructors do spend time on them in lecture. Several instructors said that they code in front of the class during lectures, and as they're going, they comment on why they're choosing to structure their code a certain way. One instructor requires CS 1410 students to attend weekly group sessions where four students get together with a TA to look at examples of student code from the most recent assignment and discuss ways the code could be written in a more elegantly or could be improved to better express the underlying ideas.

All instructors noted that this kind of feedback and instruction is needed. One instructor noted that these structure issues “... may escape the autograder. The program is correct, but it is horrible.” Another instructor indicated that even though they lecture about these topics, “... this is a problem with a large CS course with complicated programs and a huge number of students. The students are able to write bad code that works, and the test cases will say ‘Good job. You get most of the credit.’” This seems to indicate that there’s a scalability barrier facing instructors and TA’s when providing targeted feedback to students on their code structure choices.

While some instructors had better visibility than others into their students’ current mastery of these code structures and their progress, no instructors had precise, accurate visibility. When looking at the output from the instructor-facing PMD tool, one instructor was surprised when the percentage of students whose code was flagged by the tool was roughly double what they had estimated, suggesting that although instructors are aware that students are using these novice patterns, they might be underestimating how prevalent they are across the whole class. Another instructor said that they’ve surveyed their CS 1410 and CS 3505 classes about their preferences on the Simplify Boolean Returns pattern. They remarked that they were surprised to learn that half the students in CS 3505 still preferred the novice pattern. Again, this seems to show that instructors do not have an accurate sense of students’ progress with regard to style, that they have a tendency to underestimate the prevalence of these patterns, and that getting a clearer picture of what’s happening is interesting and important to them. Another instructor said that the large class sizes make it so that they do not actually look at student code, except in very rare cases, and they added that this was "sad," again suggesting that understanding what students are doing is important to instructors, and instructors think it's a problem that there's currently no scalable way to do this.

Instructors currently teach style by demonstration, and they do not evaluate students' use of code structure. They rely on TA's to use their judgement to grade and give feedback on aspects of style such as whitespace, comments, and brace placement. Scalability is a barrier to accurately perceiving student progress with regard to style.

Instructor Opinions on Style Instruction Over the Course of the Degree

One instructor reviewed the Simplify Boolean Returns pattern and remarked, "Students in all classes do this. 1410. 3500. Doesn't matter," and other instructors echoed this, saying that they see almost no improvement in how students structure their code after three semesters of programming instruction. This seems to imply that instructors want students to improve their use of code structure and to use expert patterns by the time they reach CS 3505, the fourth course in the series of major requirements, but students are not meeting these expectations. Given that instructors want to see students using expert patterns by CS 3505, at what point in the degree would they want to start giving instruction, feedback, and grades based on these patterns? One instructor who teaches CS 1410 and 2420 observed, "These habits, the longer you do them the harder it's going to be to undo them," suggesting they would prioritize addressing them for practical reasons. All instructors said that these patterns are appropriate subject matter for CS 1410.

Teaching students about these patterns and evaluating their use of them in CS

1410 comes with certain challenges. As one instructor observed, “1410 is their first semester of programming and ... they feel so triumphant just when it compiles. For you to tell them like ‘yeah, it compiles, and it works, but really you should be doing something else here…’ might be too much for them. I don't know. … But the question would be ‘Are the students comfortable enough with their own programming to really get anything out of it then?’ And I don't know the answer. I suspect some are, but some probably aren't,” suggesting that there’s uncertainty about how effective this instruction would be in CS 1410. Another instructor who also felt that code maintainability should be combined with code development from the beginning indicated that there wasn’t enough time to include it along with everything else that’s taught in CS 1410. Time came up as a limiting factor for a CS 2420 instructor as well, and despite the fact that they had previously observed that CS 2420 is not a programming class, they concluded, “... if it doesn't get cleared up in 1410 I think it's almost worthwhile to make sure it gets cleared up in 2420 and not have it persist into later classes. … maybe it's worth sacrificing not getting to cover something else.” Together, these instructors’ thoughts seem to suggest that the current curriculum is already very packed and including something new may require instructors to be willing to make tradeoffs.

Instructors want to address these code structures early in the degree because the material is appropriate to the early courses and the longer bad habits persist, the harder they are harder to break. Instructors of non-programming courses are willing to sacrifice instruction time in those courses to ensure that students build these skills early in the degree. Instructors are uncertain at what stage in the degree students would be ready to incorporate these concepts into their programming.

Do Instructors Perceive that Students Use These Patterns? Does It Matter to Them?

Of the patterns presented (Appendix: Instructor Interview Coding Patterns), all instructors said that they observed students using novice structure for the following patterns and the instructors felt that it was problematic: Simplifying Boolean Returns, Repeating Code Within an If-Block and Else-Block, Splitting Out Special Cases When the General Solution is Present, and If-Statements for Exclusive Cases. All instructors observed students using novice structure for the Collapsible If Statements pattern, but one instructor felt that in some cases there are varying opinions even among experts on how to approach this pattern. All instructors observed students using novice structure for the While-Loop When a For-Loop is More Appropriate, but one pointed out that this pattern could be language dependent and another observed that students tended to prefer whichever loop they learned first and they felt this was not a problem.

One instructor based their evaluation of the patterns partly on whether the novice-structured pattern produced inefficient assembly code. All instructors indicated that certain patterns, such Simplifying Boolean Returns, were important to them because they were more likely to cause bugs and be difficult to modify. Three instructors noted that the use of certain novice patterns was important to them because it indicated to them that the code's author had a shallow understanding of the underlying concepts. For example, repeating code within an if-block and else-block was an indication that the student did not understand which parts of the logic were actually conditional. Using if-statements for exclusive cases indicated to instructors that the student did not completely understand the enumeration and separation of the possible cases.

How Do Instructors Want to Use the Instructor-Facing Tool?

While no instructors wanted to directly use the spreadsheet of raw data output by the instructor-facing PMD tool, all of them were interested in seeing it extended. Instructors had user interface extension ideas. Three instructors wanted to see the tool incorporated as an autograder where students would lose points for using novice patterns. Two were interested in seeing a graphical summary of output, such as a pie chart displaying what percentage of students used a certain pattern, or a bar graph showing the percentage of students who had multiple flags, with the intent of using the summary to choose what to emphasize in lecture. All instructors wanted to use the tool to provide students targeted, personalized feedback by automating the detection and flagging of problematic code. Instructors also had functionality-related concerns. All instructors wanted to see a verification of the robustness and correctness of the tool Some questions they specifically wanted answered are: Does the code actually demonstrate the pattern the tool caught? Is there overlap between the tool's patterns? Are there any known cases that give false positives? Ultimately, instructors want to use the tool to better scale grading and feedback on these code structures, either through automated grading or an improved visualization of the data, and they want an empirical understanding of the reliability of the tool.

How do Instructors Want to Use the Student-Facing Tool?

Instructors also provided input on how they would want to use the student-facing tool and how they'd want to see it improved. Some instructors wanted students to have PMD freely available as an Eclipse plug-in that runs automatically at compile-time while they're coding their homework. Others envisioned only allowing students to run it on their code after they got their grades back, with a window of time where students could correct their flagged issues and resubmit to earn some points back.

One instructor is currently emphasizing positive feedback in their current rubrics and was interested in whether PMD could catch and flag positive things in student code, such as use of expert patterns. Another instructor was interested in whether PMD could catch use of inappropriate syntax, such as some unusual and unnecessarily complex code that was copy-pasted from Stack Overflow.

All instructors indicated that the tool needed to provide more explanation to students when a pattern is flagged. Some suggested linking to a website with additional details or showing side-by-side examples in Eclipse, while others suggested showing an embedded video or making the student take a quiz to demonstrate their understanding of the pattern. Instructors also emphasized that the phrasing of the feedback would not be clear to a student who is using these patterns in the first place, and one instructor suggested rephrasing the feedback in a way that was actionable or making it suggest to the student that they think carefully and reconsider their code structure choice. Instructors also wanted to give students a way to know they had preserved the logic of their original code if they made changes based on the tool's feedback.

Instructors want the tool to give more comprehensive, instructive feedback to the student, including positive feedback for use of expert code structures and assurance that the code's logic was unchanged as a result of the structure revision. Three instructors wanted students to have the tool while they coded, and one instructor wanted students to only have access to the tool after they submitted their assignment.

INTERVIEW DISCUSSION

Using the Tool to Address Scalability Issues

Current style grading and feedback processes do not scale well to large class sizes, and so instructors aren't able to assess and target style effectively. The instructor-facing tool has the ability to provide instructors detailed insight into students' progress and persistent or prevalent gaps in the class's command of specific code structures. No instructors want to interface with the output of the tool the way it currently is presented, but all of them were interested in using it to get an understanding of which patterns students are struggling with so they could adapt their instruction. Incorporating a graphical user-interface (GUI) that visualizes the data the tool already outputs as a bar graph or pie chart will be an important next step, and it will be an easy improvement to make. These graphs would convey statistical information that instructors are interested in, such as the number of occurrences of each novice pattern across all assignments or the percentage of students that used a given novice pattern. Another round of feedback from instructors would be valuable once an initial prototype of the GUI is ready.

Another shortcoming in the current style feedback process is that there's a delay of a week or more between students composing the code and receiving feedback. Instructors indicated that this makes it difficult to hold students accountable on the following assignments for having learned from style mistakes on previous assignments. To address this, some instructors indicated that they would want students to have access to the student-facing tool throughout their coding process, and others indicated that they would want students to be able to use the tool to correct mistakes and earn back points on their assignment only after it had been graded for style. Withholding the student-facing tool will prevent it from masking gaps in students' command of style, but the approach the instructor suggested would require a round of regrading. More investigation needs to be done to understand whether this visibility into students' command of style is the only reason instructors would want to withhold the tool. If this is the only reason, perhaps other approaches, such as logging student revisions and the tool's feedback, would provide that insight without requiring a second round of grading.

If the tool can be shown to be reliable, instructors would want to use it as part of the grading process. They said they would use it as an autograder because they believe that students will make changes to the way they code if it means they'll get more points. Instructors would also want to use it to help TA's provide students more targeted feedback. All instructors were concerned with the current logistics of having TA's give detailed feedback. They explained that asking TA's to style feedback historically has not made much of a difference partly because the time it takes to leave detailed feedback doesn't scale well in large classes and the variance in judgment among the TA's when reading code. Extending the tool so that its output works with Gradescope autograding and in-line comment feature would be a valuable next step in making this tool meet instructors' needs.

Tool Design Recommendations

Instructors want to know if the tool is reliable before they incorporate it into their teaching, feedback, and evaluation workflow, so investigating and reporting the accuracy of the tool will be an important next step toward instructors using the tool. At this point in development, there have been cases of duplication of feedback for Collapsible If Statements; when this pattern is caught by multiple PMD rules, it is flagged multiple times, giving an inflated impression of the prevalence of this pattern (see Figures 2 and 3). Some versions of the Simplify Boolean Return and Collapsible If Statement patterns are not detected currently (see Tables 1 and 2), such as non-idiomatic use of the ternary operator, leading to false negatives. To evaluate the tool's reliability, its output and the code it was run on need to be inspected by hand for false negatives, false positives, and duplicate feedback.

Next Research Steps and Instructional Design Recommendations

The instructors' perception that the Repeating Code Within an If-Block and Else-Block, Splitting Out Special Cases When the General Solution is Present, and If-Statements for Exclusive Cases patterns are entangled with the student's understanding of the problem space and the flow of logic suggests that there's another layer to teaching and learning about code structure that needs to be accounted for when designing the student-facing tool. Investigating this theory is an important next step because if this theory is accurate, it will mean that instructors will need to take extra steps to determine and address the root cause of the issue.

Instructors want students to learn style and maintainability starting in their very first programming class, but instructors also expressed uncertainty about whether CS 1410 students would be “ready” to learn to integrate expert patterns effectively in their programming. Therefore, a next research step will be to investigate what practices and concepts CS 1410 and CS 2420 students learn and retain from the tool. In pursuit of that research question, the tool would need to be adapted to log the writing and revising steps the author goes through and the feedback they are presented at each step.

As stated above, instructors are unsure if students will be comfortable enough with their own programming in CS 1410 to learn from the student-facing tool. One CS 2420 instructor indicated CS 2420 isn’t a programming class, and so code structure isn’t emphasized there currently. However, that same instructor said that it’s so important to address these novice code patterns that they would consider devoting time to it in CS 2420 to make sure these problems don’t persist beyond that point. If most CS 1410 students aren’t at a point where the tool can make a measurable difference in their command of expert patterns, and there isn’t time to adequately address these patterns in CS 2420, then the curriculum needs to be adapted.

One CS 1410 instructor interviewed has recently started to incorporate weekly, TA-guided code review sessions where four students spend roughly ten minutes discussing ways to improve a provided code sample, taken from an anonymous submission to the most recent homework assignment. Research shows that pedagogical code reviews such as this in a first-semester CS course context can lead to improved student code quality, a stronger sense of community, and increasingly sophisticated discussions of programming issues and practices [4]. This process should be adopted universally in all offerings of CS 1410. The pedagogical code review designed by

Hundhausen et al. [4] was staged over the course of a semester into 3 sessions, each 170 minutes long and consisting of 21 participants. This model may be too resource intensive and may need to be adapted to fit in the context of CS 1410, which already has a demanding, weekly lab component. The rules used by PMD can be incorporated into the checklist of best practices that students use to examine each other’s code. Introducing first- or second-semester students to code patterns and style in this guided and structured way lays a strong foundation that can then be expanded on in CS 2420.

Rather than removing material from CS 2420 and replacing it with these topics, a one credit-hour co-requisite for CS 2420 should be introduced. Skills and topics covered and practiced in this class would then be reinforced by grading and feedback in CS 2420. One implementation of this kind of class [15] found that the quality control of the feedback given to students and the cost of running the course was a concern when transitioning to a larger scale (100 students). PMD could be extended to address this concern, improving the student experience and easing the burden on instructors. Programming exercises specifically targeting the relevant code structures would need to be developed, and the tool would need to be able to track and report student progress. These would be substantial tasks. The student-facing tool doesn’t currently record any data, such as error messages, file snapshots, or keystrokes between runs.

COMBINED DISCUSSION

Tool and Instructional Design Recommendations

When students wrote code that used expert patterns during the think-alouds, PMD gave no feedback, and this was confusing to students. One instructor indicated that they want students to receive specific, positive feedback on things they did well in their assignments. Extending PMD so that it flags expert patterns and provides positive feedback on them would help address this usability issue students encountered, it would help students recognize when they've used expert structure, and it would provide a feature that an instructor wants.

When PMD flagged novice patterns during the think-aloud, students did not consistently interpret the feedback correctly. All instructors expressed concern that the current PMD feedback is insufficient for teaching students why and how they should use expert patterns. For example, the felt that highlighting an instance of the Simplify Boolean Returns pattern in the student's code and telling them to "Avoid unnecessary if..then..else statements," would not effectively convey a motivation or explanation for coding a different way, especially to someone who is using that pattern in the first place. Three of the four instructors also expressed concerns that the feedback was not worded in an actionable way. PMD's feedback needs to be revised to be more actionable and to provide information that would be significant and useful to a student who is learning how and why to use these patterns.

When revising their code for style, all students but one were clearly aware of the correctness implications of their revisions and did not blindly copy and paste code. Still, most students expressed uncertainty about whether their revision did the same thing as their original code. One instructor pointed out that when students come for help with their code, and their code is needlessly complex and difficult to understand and change, the students are extremely averse to suggestions to modify their code to improve its readability or maintainability. The instructor theorized that this is because the students had already put so much work into the code that they didn't want to spend time fixing something that (probably) wasn't broken and risk introducing new problems. Extending the tool so it could determine and convey whether the student's change preserved the original logic would help address this resistance and hesitancy. This extension would be difficult. Deciding when this feature runs and incorporating its functionality into the existing tool logic is one concern, but in addition to this, determining the semantics of a piece of code at compile time is difficult (more difficult for some patterns than others).

Next Research Steps

Students' word choices while thinking out loud raised questions about how best to phrase the tool's feedback. While completing the `isStrictlyPositive` task, Participant 4 used the term "statement" correctly at times (e.g. "make it a single line statement") and incorrectly at other times (e.g. "for a return type where they actually make it a Boolean statement itself"). This is interesting because it suggests either the student knows the correct terms (and just is using the wrong terms by mistake or by lack of attention to detail) or the student has an incorrect understanding of the difference between a statement and an expression. This distinction will affect the wording of the PMD feedback. This student misinterpreted the current feedback, but would they know what to do if it instead said something like, “Return the Boolean expression directly”? Or would that lead to more confusion if they're not confident about what an expression is? When investigating the effectiveness of error messaging in DrRacket, Marceau et al. [8]noted from interviews with students that:

> [S]tudents misused words, or used long and inaccurate phrases instead of using the precise technical terms when describing code,” and “…some exchanges during the interview suggested that the students' poor command of the vocabulary undermined their ability to respond to the messages (Marceau et al., 2011, p. 3).

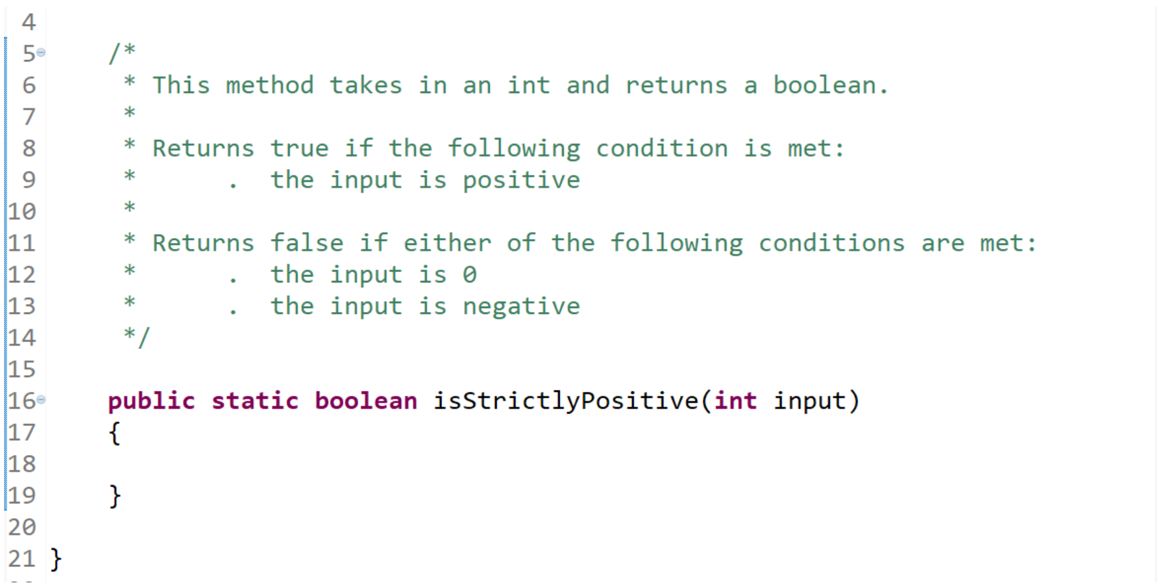
Understanding how this finding applies to student interactions with PMD also tie in with instructor concerns about what course in the degree would be most appropriate for students to use the tool. The level of vocabulary used in the tool’s feedback needs to match the level of vocabulary that the student has learned.

Another dimension of the feedback is its intent. All instructors interviewed want the PMD feedback to be more actionable. However, Marceau et al [8] specifically recommended against error messages that propose a solution in the case where the suggested fix might not cover all cases. Testing whether or not actionable feedback from PMD can always be applied to the flagged code needs to be included as part of investigating the reliability of the tool and refining the phrasing of the feedback.

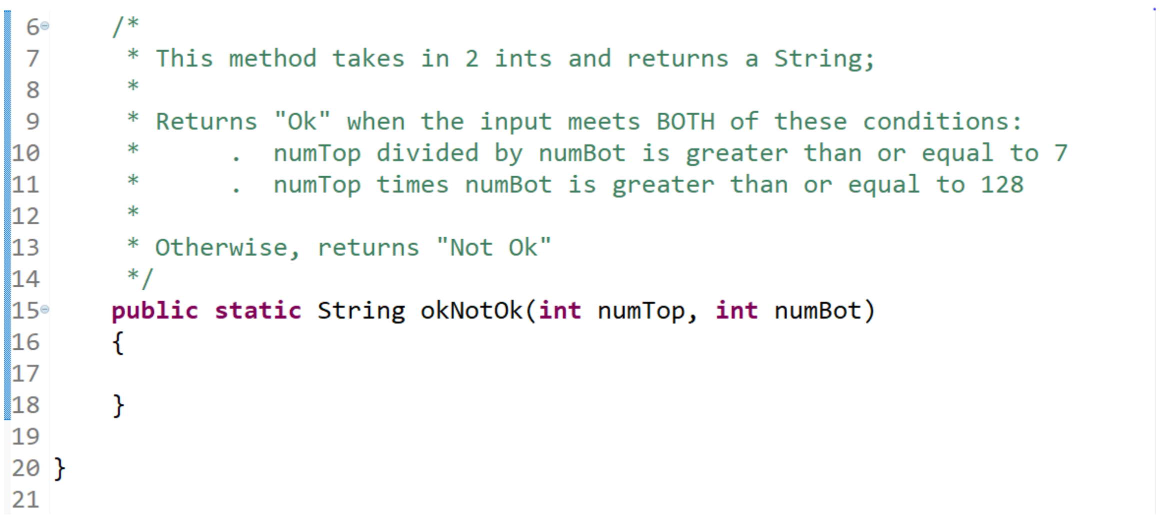
CONCLUSION

PMD flagged the code structures it was expected to, and students had some success at interpreting its feedback and revising their code structures. PMD’s feedback still needs reworking to better facilitate learning. Instructors want to incorporate the student-facing tool and batch-processing tool in their workflow, but they want more detailed information about the tool's reliability and the output of the instructor-facing tool needs to be made user-friendly.

APPENDIX: THINK-ALOUD TASKS



```java

    /*
     * This method takes in an int and returns a boolean.
     *
     * Returns true if the following condition is met:
     *      .  the input is positive
     *
     * Returns false if either of the following conditions are met:
     *      .  the input is 0
     *      .  the input is negative
     */

    public static boolean isStrictlyPositive(int input)
    {

    }

}
```



```java
    /*
     * This method takes in 2 ints and returns a String;
     *
     * Returns "Ok" when the input meets BOTH of these conditions:
     *      .  numTop divided by numBot is greater than or equal to 7
     *      .  numTop times numBot is greater than or equal to 128
     *
     * Otherwise, returns "Not Ok"
     */
    public static String okNotOk(int numTop, int numBot)
    {

    }

}

```

APPENDIX: CODING PATTERNS

Simplify Boolean Returns Pattern

Novice Version 1

```
if (i == 1) {
     return true;
} else {
     return false;
}
```

Novice Version 2

```
boolean isNull = false;
if (head == null) {
     isNull = true;
}
```

Novice Version 3

```
boolean b;
if (i == 1) {
     b = true;
} else {
     b = false;
}
return b;
```

Novice Version 4

```
return i == 1 ? true : false;
```

Expert Version

```
return i == 1;
```

Collapsible If Statements Pattern

Novice Version 1

```
if (num != 0) {
     if (num != 1) {
          return “Ok”;
     }
     return “Not Ok”;
}
return “Not Ok”;
```

Novice Version 2

```
if (min < num) {
     return max > 1;
}
return false;
```

Novice Version 3

```
if (min == 0)  {
     return “Not Ok”;
}
if (max/min > 0) {
     return “Ok”;
}
return “Not Ok”;
```

Expert Version 1

```
return min < num && max > 1;
```

Expert Version 2

```
if(min != 0 && max/min > 0) {
     return “Ok”;
}
return “Not Ok”;
```

While Loop Should Be For Loop Pattern

Novice Version 1

```
int i = 2;
while (i < array.length) {
  array[i] = array[i] + 2;
  i++;
}
return array;
```

Expert Version

```
for (int i = 2; i < array.length; i++) {
  array[i] = array[i] + 2;
}
return array;
```

Repeating Code Within an If-block and Else-block Pattern

Novice Version 1

```
String name;
double price);
double discountPrice;
double over50Discount = .2;
double discount = .1;
if (price > 50) {
  discountPrice = price * (1 - over50Discount);
  return "The original price of " + name + " was $" + price
           + " but you get a discount of " + over50Discount
           + " so you only pay $" + discountPrice;
} else {
  discountPrice = price * (1 - discount);
  return "The original price of " + name + " was $" + price
           + " but you get a discount of " + discount + "
so you only pay $"
           + discountPrice;
}
```

Expert Version

```
double sale = 0.25;

double specialSale = 0.5;

double over50Sale = 0.35;

double actualSale = 0;


String beginning = "Your item, " + item + ", usually costs
" + price + ", but ";

String ending = "you are getting it for ";


if(item.equals("socks") && coupon) {

     beginning += "since you have a coupon, ";

     actualSale = specialSale;

}
```

```
else if (price > 50) {
     beginning += "since it is over 50, ";
     actualSale = over50Sale;
}
else
     actualSale = sale;

String realEnding = "Congratulations!! you saved ";
double savings = price - price * (1-actualSale);
realEnding += savings + "!";
return beginning + ending + price * (1 - actualSale) +
realEnding;
```

Splitting out a Special Case When the General Solution is Present Pattern

Novice Version 1

```
if (first.length() != second.length()) {
  return first + " and " + second + " do not have the same
  letters.";
}

char letters1[] = first.toCharArray();
char letters2[] = second.toCharArray();
Arrays.sort(letters1);
Arrays.sort(letters2);
if (Arrays.equals(letters1, letters2)) {
  return first + " has the same letters as " + second;
 }
return first + " and " + second + " do not have the same
  letters.";
```

Expert Version

```
char letters1[] = first.toCharArray();
char letters2[] = second.toCharArray();
Arrays.sort(letters1);
Arrays.sort(letters2);
if (Arrays.equals(letters1, letters2)) {
  return first + " has the same letters as " + second;
 }
return first + " and " + second + " do not have the same
  letters.";
```

If-Statements for Exclusive Cases (Rather Than If-Else) Pattern

Novice Version 1

```
String size = "";
if (i < 10) {
  size = "small";
}
if (i >= 10 && i < 20) {
  size = "medium";
}
if (i >= 20) {
  size = "big";
}
return "The size is " + size;
}
```

Expert Version

```
String size = "";
if (i < 10) {
  size = "small";
}
else if (i >= 10 && i < 20) {
  size = "medium";
}
else {
  size = "big";
}
return "The size is " + size;
```

REFERENCES

[1] Hannah Blau and J. Eliot B. Moss. 2015. 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*, ACM, 15–20. DOI:https://doi.org/10.1145/2729094.2742622

[2] Chris Dahlberg. 2016. StyleCop. Retrieved from https://marketplace.visualstudio.com/items?itemName=ChrisDahlberg.StyleCop

[3] Robert L Glass. 2002. *Facts and Fallacies of Software Engineering*. Addison-Wesley, Boston. DOI:https://doi.org/http://dx.doi.org/10.1109/FOSE.2007.29

[4] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating Pedagogical Code Reviews into a CS 1 Course: An Empirical Study. *Proc. 40th ACM Tech. Symp. Comput. Sci. Educ.* (2009), 291–295. DOI:https://doi.org/10.1145/1508865.1508972

[5] InfoEther. 2002. PMD. Retrieved June 26, 2019 from https://pmd.github.io/

[6] Saj-NIcole A Joni and Elliot Soloway. 1986. But My Program Runs! Discourse Rules for Novice Programmers. *J. Educ. Comput. Res.* 2, 1 (1986), 95–125.

[7] Jin-Su Lim, Jeong-Hoon Ji, Yun-Jung Lee, and Gyun Woo. 2011. Style Avatar: A Visualization System for Teaching C Coding Style. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (SAC '11), ACM, New York, NY, USA, 1210–1211. DOI:https://doi.org/10.1145/1982185.1982451

[8] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind your language: On novices'interactions with error messages. In *ONWARD!'11 - Proceedings of the 10th ACM Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.

[9] Matthew Peveler, Jeramey Tyler, Samuel Breese, Barbara Cutler, and Ana Milanova. 2017. 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*, 641–641. DOI:https://doi.org/10.1145/3017680.3022384

[10] Hannah Potter. 2019. Personal Communication.

[11] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient

elements in novice solutions to code writing problems. *Conf. Res. Pract. Inf. Technol. Ser.* 114, (2011), 37–45.

[12] Eliane S Wiese, Anna N Rafferty, and Armando Fox. 2019. Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated. In *Proceedings of the 41st International Conference on Software Engineering*, ACM, 84–94.

[13] Eliane S Wiese, Anna N Rafferty, Daniel M Kopta, and Jacqulyn M Anderson. 2019. Replicating Novices' Struggles with Coding Style. In *Proceedings of the 29th International Conference on Program Comprehension*, ACM, 13–18.

[14] Eliane S Wiese, Michael Yen, Antares Chen, Lucas A Santos, and Armando Fox. 2017. *Teaching Students to Recognize and Implement Good Coding Style*. ACM, Cambridge, MA.

[15] M Woodley and S N Kamin. 2007. Programming studio: A course for improving programming skills in undergraduates. *SIGCSE 2007 38th SIGCSE Tech. Symp. Comput. Sci. Educ.* January 2007 (2007), 531–535. DOI:https://doi.org/10.1145/1227504.1227490