Use of an Anti-Pattern in CS2: Sequential `if` Statements with Exclusive Conditions

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

Matthew Hooper
University of Utah
Salt Lake City, USA
matthew.hooper@utah.edu

Adriana Salazar
University of Utah
Salt Lake City, USA
u1328390@utah.edu

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

ABSTRACT

How can we teach students to use more readable code structures? How common is it for students to choose less readable (but still functional) alternatives? We explore these questions for a specific anti-pattern: using sequential `if` statements when conditions are exclusive (rather than using `else-if` or `else`). We created and validated an automated detector to identify this anti-pattern in student's code. Running the detector on 1,764 homework submissions (from 270 students in a CS2 class on data structures and algorithms) showed that this anti-pattern was common and varied by assignment: across 12 assignments, 3% to 50% of submissions used sequential `if`s for exclusive cases. However, using this anti-pattern did not preclude using `else-if`s : for one assignment, 34% of submissions used both forms. Further, students used sequential `if` statements in surprising ways, such as checking a condition and then the negation of that condition, indicating a more novice level of understanding than expected for an intermediate course. Hand-inspection of the detector-flagged cases suggests that sequential `if`s for exclusive cases may be a code smell that can indicate larger problems with logic and abstraction.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **Software and its engineering** → **Patterns**.

KEYWORDS

Code structure, Discourse rules, Static code analyzer, Anti-patterns

ACM Reference Format:
Sara Nurollahian, Matthew Hooper, Adriana Salazar, and Eliane Wiese. 2023. 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 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569744

1 INTRODUCTION: CODE STRUCTURE

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569744

```
if(midpoint - item > 0){
   end = midpoint - 1;
}
if(midpoint - item < 0){
   start = midpoint + 1;
}
if(midpoint - item) == 0){
   return true;
}
```

```
if(midpoint - item > 0){
   end = midpoint - 1;
} else if(midpoint - item < 0){
   start = midpoint + 1;
} else{
   return true;
}
```

Table 1: Left: the *exclusive ifs* anti-pattern, using `if` statements with exclusive conditions. Right: the desired pattern, using `else-if` and `else` to make the exclusivity explicit.

Well-structured, idiomatic code is often called "clean" [32], "elegant", and "beautiful" [33]. Such code communicates the programmer's intentions and is easier for others to read, modify, and maintain [21]. However, teaching students to write well-structured code is hard, partially because experts' knowledge of code structure is often implicit [37]. Studies measuring the extent of code structure issues in student code motivate further research and instruction on those anti-patterns [19, 27]. Here, we explore the prevalence of the exclusive ifs anti-pattern [41]: checking exclusive conditions with a sequence of `if`-statements rather than `else-if` or `else` (Table 1).

While many anti-patterns relate to structures taught in introductory classes (e.g., `if`-statements, loops), inappropriate structure choices persist across all levels of undergraduate courses [15, 24]. Instructors may not grade student work on code structure, understandably focusing on metrics that can be auto-graded, such as passing test cases [35]. While hand-inspection of code and in-person code reviews are useful, they are time intensive [23, 44]. Creating automated tools to detect anti-patterns offers scalability and could provide students with immediate feedback while coding.

We explore the *exclusive ifs* anti-pattern in CS2 students' homework assignments. This anti-pattern was identified in prior work [41] and is important to instructors at our institution. We built a detector for *exclusive ifs* and used it to answer our first 2 research questions:

- RQ1 How common are novice patterns in the wild? Specifically, the exclusive `if`s anti-pattern on CS2 homework?
- RQ2 What sub-categories of the exclusive `if`s anti-pattern or co-occurring anti-patterns do students demonstrate?
- RQ3 How do learning resources teach this structure?

2 PRIOR WORK: STUDENT ANTI-PATTERNS

Code quality encompasses many facets, from white space and variable names to control flow structural choices. We examine a specific control flow as an anti-pattern, an approach that is distinct from summarizing code quality in a metric. Some anti-pattern research creates detectors for specific assignments, relying on a corpus of prior student submissions (e.g., [43]). We focus on strategies that can apply across assignments. This type of anti-pattern research often takes one of two approaches. The first is identifying anti-patterns and their frequencies at scale, using professional code analyzers (e.g., PMD [12], CheckStyle [10], Clang Static Analyzer [1]) that perform static analysis on the code's Abstract Syntax Tree (AST) [17, 19, 26, 27]. However, such analyzers are not sufficient for students to improve their own code [27], and this research approach is inherently limited to the patterns that can be detected on an AST. The second approach is to describe important anti-patterns [28, 40, 41] and propose ways to teach students to revise or avoid them [29, 39]. However, the more-complex patterns described in this line of work cannot be identified from an AST alone; unsurprisingly, we lack detectors for them. Most research on student code structure does not involve building custom detectors for anti-patterns that are missing in professional code analyzers (de Ruvo et al. [16] and Ureel et al. [38] are exceptions, detecting anti-patterns from the AST). Additionally, research on anti-patterns typically does not look for *correct* pattern usage. Drawing on the strengths and limitations of this prior work, we explicate an important anti-pattern, build a detector for it, and examine students' use of the anti-pattern and pattern. We are not aware of prior work examining students' use of *exclusive ifs* in code writing. Keuning et al. built a tutoring system to teach a subset of this pattern (using `else` rather than `if(a)-if(!a)`) [29], and prior work has examined related anti-patterns (nested if-statements rather than conjoined conditions [42]; explicit `if-else` [16, 29] rather than directly returning a Boolean expression).

3 THE EXCLUSIVE IFS ANTI-PATTERN

The *exclusive ifs* anti-pattern is a set of sequential `if`-statements that check mutually exclusive conditions (Table 1, Left). Exclusive conditions should use `else-if` to make the exclusivity explicit (ending with an `else` when the conditions cover all possibilities exhaustively). However, since some instructors see control transfer statements as communicating exclusivity [41], we do not apply *exclusive ifs* to those cases (discussed in section 4.2). Replacing *exclusive ifs* with `else-if` statements will not change the code's functionality. However, `else-if` and `else` statements make the programmer's intent clearer. With sequential `if`-statements, a reader must examine the intersections between the conditions to determine what inputs cause multiple `if`-statement bodies to execute. The correct pattern can save a reader the trouble of determining that there are no such intersections and can make the code easier to maintain and modify by explicitly showing which conditions are related. However, early programmers may prefer sequential `if` statements since all the conditions are explicit. Our algorithm for detecting instances of this anti-pattern is in Section 4.2. All examples in this paper of code flagged by the detector are from student homework submissions (some lightly edited for length).

4 METHODS

We built and validated a detector for exclusive `if`s and used it to analyze the programming homework submissions from an offering of a CS2 course (on data structures and algorithms) of a large public university in the U.S. This analysis was performed on anonymized assignments after the course was over, under exempt *IRB*_00123431.

4.1 Homework Data and Classroom Context

Data structures and algorithms (CS2) is a required course for CS minors and majors. As introductory programming (CS1) is a prerequisite, all students in the CS2 course are familiar with programming and conditional statements. Further, CS1 includes lecture examples and personalized feedback on the use of some anti-patterns, including exclusive `if`s. In CS1 and CS2, code structure rarely impacts grades. However, in the subsequent software course, 35% of assignment grades are based on code structure, style, and design.

CS2 includes 12 Java homework assignments, worth 50% of the course grade, of which code quality is 5% (2.5% of the final grade). Code quality is based on commenting, variable names, and consistent formatting (e.g., brackets, white space). Code structure patterns are not part of the code quality rubric. Hooper, a former teaching assistant (TA) for this class, reports that TAs grade the homework and do not look for quality issues beyond the rubric. Also, during help hours, students focus on getting their code to function correctly, with little (if any) time spent on code quality. However, if instructors notice such issues, they bring them up in class discussions. The 12 assignments are primarily graded on functionality, the exhaustiveness of students' test cases, and Big O performance. Students are shown some tests and can run the auto-grader on them before the deadline. After the deadline, assignments are auto-graded against the provided test cases, and additional hidden ones. Our data includes only the final submission for each assignment and provides a baseline view of exclusive `if`s from students who were not incentivized to attend to this anti-pattern.

Assignment 1 was done individually (267 submissions) and the rest were done in pairs (131 to 142 submissions). Since students could drop one homework grade, they did not submit all assignments. Overall, we have submissions from 270 students. As students changed partners over the semester, we cannot analyze a pair's progression over the course. We explain the details of the assignments important to our findings in sections 5.1 and 6.

4.2 Exclusive Ifs Detector

We developed a detector to identify *exclusive ifs*. The detector is an extension to PMD [12], an open-source code analyzer for Java and other languages. PMD compiles the code into an AST and examines it for anti-patterns. While we can identify a pair of sequential `if` statements from the AST alone, deciding on exclusivity requires checking the logic of the conditions. Thus, to check exclusivity, we use Z3 [14]; a Satisfiability Modulo Theories (SMT) solver.

The exclusivity test can be framed as a Boolean satisfiability problem. Consider `p` as a Boolean value. Since no value for `p` makes `(p && !p)` true, it is unsatisfiable: `if(p)-if(!p)` are *exclusive ifs*. Z3 extends Boolean satisfiability to a variety of data types, including numbers, arrays, and strings. We developed a translator to use Z3 to test the conditions of `if` statements written in Java. Z3 handles

Java expressions with primitive type. It translates expressions with Object type as Int, corresponding to the Object address in memory, preserving the behavior of the "==" operator for objects. Our detector traverses the AST until it finds a sequential pair of `if` statements. Using Z3, it checks the conjunction of the conditions, and when the conjunction is unsatisfiable, it flags the second `if`. However, there are two exceptions that we do not test for exclusivity: (1) a pair of `if` statements where either `if` has an `else` (Examples 1 and 2 in Listing 1); and (2) when the body of first `if` has a transfer control statement (since that imposes exclusivity, as in Example 3 in Listing 1). In Java, five statements transfer control: `return`, `break`, `continue`, `throw`, and `yield`.

```
Example 1: if-statements are sequential in the AST but not in the code.
if (x < 0) message = "negative";
else message = "not negative";
if (x > 0 && x < 1) message += " tiny positive";

Example 2: The second if statement has an else child.
if (y > 10) message = "Greater than ten.";
if (y < 10 && y % 2 == 0 ) message = "Even and smaller than 10.";
else message+= " An odd number or greater than or equal to ten.";

Example 3: The first if statement has a return statement.
if (z > 0) {
    message="greater than zero";
    return;
}
if (z < 0) message="less than zero";

Example 4: The second if is exclusive with the first, but not its else-if child
if ( x % 4 == 0) message = "Multiple of 4.";
else if ( x % 4 == 1) message = "1 more than a multiple of 4.";
if (x % 2 == 1) message + = " An odd number.";
```

Listing 1: Sequential AST `if` statements that are not flagged.

Also, if a candidate pair of `if` statements has `else-if` children, the second `if` is only flagged as a violation if the conditions of the `if` statements are exclusive with the other's children (example 4 in Listing 1 would not be flagged). Algorithm 1 describes our detector, available on github[1]. To the best of our knowledge, this is the first open-source detector for the exclusive `if`s anti-pattern.

The detector has two main limitations. First, it identifies `if`s that should be `else-if`s or `else`s, not `else-if`s that should be `else`s. Second, it does not consider side effects. When the conditions call methods, the detector assumes a method will have a consistent result if (and only if) it is called with the same arguments. However, in reality, the same method with the same arguments can have different results. In Example 1 in Listing 2, flagged during development, the first call to `checkin()` returns true if the book was successfully checked in to the library, and, due to a side effect, the second call returns false (a checked-in book cannot be checked in again). Further, the same method with different arguments can have the same results, as in Example 2 in Listing 2, which was not flagged by the detector.

```
Example 1: These ifs are not exclusive because checkin() has a side effect
if(!lib2.checkin(9781843190004L))
    System.err.println("TEST FAILED -- library: check in");
if(lib2.checkin(9781843190004L))
    System.err.println("TEST FAILED -- library: check in again");

Example 2: These ifs are exclusive even though the method arguments differ
if (aChar.equals('/'))
    isComment = true;
if (aChar.equals('*'))
    isMultiLineComment = true;
```

Listing 2: A false positive (Ex. 1) and false negative (Ex. 2)

[1]https://github.com/elianeswiese/code_structure_pattern_detector

Algorithm 1 Detecting exclusive ifs

for consecutive pairs of if statements (**first**, **second**) **do**
  if first contains transfer control statements **then**
    continue
  if first ends with `else` or **second** ends with `else` **then**
    continue
  if first is exclusive with **second** and each of **second**'s `else if` children **then**
    if second is exclusive with each of **first**'s `else if` children **then** addViolation (**second**)

4.2.1 Validating the Detector. Before developing the detector, we divided the submissions into training and test sets (for developing and evaluating, respectively). For the test set, we randomly chose 15 students who submitted all assignments, representing 10% of the total number of submissions. The rest of the data was the training set. For each assignment, we randomly selected 2-3 submissions from the training set and hand-inspected them, identifying 156 pairs of sequential `if` statements, which were further examined for exclusivity. We ran the detector on these samples, examining detector-human disagreements to improve the detector. The detector's final recall and precision on these samples from the training set were 0.97 and 0.95. To validate the detector, Salazar, (who was not involved in building the detector), hand-inspected every `if` statement in the test set. Comparing the detector's performance on the test set to this human ground-truth (Table 2) results in a recall of 0.85, precision of 1, Cohen's k of 0.906, and 98.20% agreement with the human. Afterward, we modified the anti-pattern detector to identify the *correct* usage of the pattern (i.e., using `else-if` with exclusive conditions). Given the performance of the original detector, we expect this modified detector to also be reliable.

Detector verification	Flagged exclusive	Not flagged exclusive
Human agreed	77	688
Human disagreed	0	14

Table 2: Validation on the test set (15*12 submissions)

5 FINDINGS

We report findings from running the detector on the entire dataset. Students frequently use the exclusive `if`s anti-pattern, doing so for 30% of all instances when exclusive conditions were checked. However, the rate varies by assignment. Many submissions include both instances of `if-if` and `if-else-if`, suggesting that students can use both, but may not know when to use which. Hand inspection revealed sub-categories of exclusive `if`s, and co-occurrence of this anti-pattern with more serious structural problems.

5.1 RQ1: Students Frequently Use Exclusive `if`s

Students' use of exclusive `if`s varied by assignment, from 3% of submissions using this anti-pattern on assignment 1 to 50% on assignment 7 (Fig. 1). For three assignments (7, 8 and 9), over 25% of submissions include exclusive `if`s (347 instances of the anti-pattern

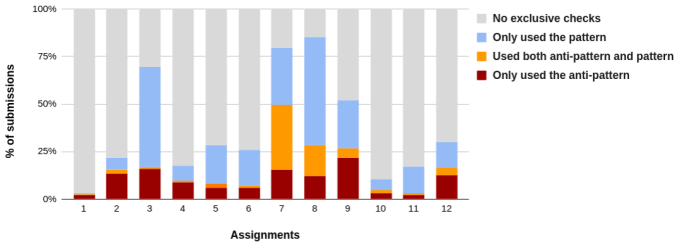

Figure 1: Percentage of submissions that include the anti-pattern, pattern, or do not check for exclusive cases at all. The sum of the red (only using the anti-pattern) and the orange (using both the pattern and anti-pattern) yield the total percentage of submissions that include an anti-pattern (e.g., 50% on #7).

over 409 submissions). Of submissions that used the anti-pattern, the number of instances per submission ranged from a mean of 1.18 on assignment #3 to 3.19 on #7. While these means are small, this range represents 15-91% of the instances when exclusive cases are checked: when examining exclusive cases, students frequently use exclusive ifs. The decision to check exclusive conditions depends on students' approaches to the problems and the different requirements of each assignment. For example, #7, "Balanced Symbol Checker," checked Java source code for unmatched parentheses, brackets, braces, and block comments. As Java programs can include non-code text (e.g., comments, strings, and escape sequences), it is necessary to account for special cases while parsing. Many students checked exclusive cases sequentially to determine if a character was a specific opening or closing symbol. In contrast, #1, "Matrix" required the implementation of 2D matrix operations like addition and multiplication, and did not necessitate checking exclusive conditions sequentially. Further, all assignments had at least one submission including both the novice and expert versions of this pattern (from 1 submission for assignment #1 to 47 submissions (34%) for #7). Within those 47 submissions for #7, 139 instances of checking exclusive cases used the anti-pattern, and 190 used the `else-if` pattern. Overall, 142 unique students submitted at least one assignment with both the pattern and anti-pattern.

For each assignment, students created files for testing the correctness of their code and for measuring its run-time with different inputs. We hypothesized that students may have approached the readability of their timing and testing code differently from the readability of their "regular" assignment code. However, only the first two assignments had more occurrences of exclusive `if`s in the testing/timing files than in the regular files.

5.2 RQ2: Sub-Categories by Required Revision

Hand-inspection of code flagged by the detector revealed different contexts where students used exclusive `if`s, requiring different revisions. Making these categories explicit may aid instructional design, as students who only see examples from one category may not transfer to the others. While we can recommend easy fixes for some categories, this is not the case for code with larger problems.

5.2.1 Replace with else-if. When the exclusive conditions do not exhaustively cover all possibilities, as in Example 1 in Listing 3, the latter `if`-statements should be replaced with `else-if`. This simple revision does not require changing the conditions. However, this type of revision can allow for simplifying the conditions in some cases (e.g., `if(x<10)...if(x>=10 && x<20)` could be simplified to `if(x<10)...else if(x<20)`.

5.2.2 Replace with else-if and else. When the exclusive conditions exhaustively cover all possibilities (as in Table 1), the middle `if`-statements should be replaced with `else-if`, and the last should be replaced with an `else`. Hand-inspection uncovered instances of using sequential `else-if` rather than ending with an `else`, as in Listing 3, Example 2 (note that this usage of `else-if` is not flagged by our detector). While straightforward replacement with `else-if` and `else` would communicate how the conditions are related (and that they are exhaustive), it would not address the ordering of the conditions (e.g., in Listing 3 Example 2, some instructors would prefer the `if`-statement with the `return` comes first).

```
Example 1: Line 5 should be else-if: conditions are exclusive but not exhaustive
if(start.charAt(i) == '<') {
    location[0] = i;
}
if(start.charAt(i) == '>') {
  location[1] = i+1;
}
Example 2: The last else-if should be an else: conditions are exhaustive
if(comparison < 0) {
  low = mid+1;
}
else if(comparison > 0) {
  high = mid-1;
}
else if(comparison == 0) {
  return true;
}
```

Listing 3: Easy-to-fix anti-patterns.

5.2.3 Replace with else. A specific case of exhaustiveness is using sequential `if`s to check a condition and its negation (Listing 4). The use of this anti-pattern in CS2 is concerning, as it suggests a lack of fluency with `else`. While our data cannot tell us *why* students used this anti-pattern, or how it is related to conceptual understanding, we consider this anti-pattern important enough to merit its own category. Note that for the example in Listing 4, this anti-pattern co-occurs with a larger problem of a helper method doing more than one thing, which we discuss in section 5.2.5.

```
public boolean CompatibleSize(Matrix m,boolean
     isMultiplication) {
    if (!isMultiplication)
      if (!(m.data.length == this.data.length) || !(m.data
     [0].length == this.data[0].length))
            return false;
    if (isMultiplication)
      if (!(this.data[0].length == m.data.length))
            return false;
    return true;
  }
```

Listing 4: Using exclusive ifs to check a condition and its negation (rather than `if-else`).

5.2.4 Replace with Switch. Another specific case is when exclusive `if`s (that are also exhaustive) could be converted to a switch case. Fitting this subcategory requires 3 characteristics: (1) the conditions only check different values of the same variable; (2) at least 4 possible values are checked; and (3) each `if`-statement body has at most two lines of code. Listing 5 shows an example: (1) each `if`-statement checks a value for the variable `maze[i][j].nodeType`; (2) it checks

for 5 different values; and (3) each `if` body has one line of code. This subcategory may indicate student difficulties in identifying when to use a `switch-case` or lack of comfort in implementing it.

```
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        if (maze[i][j].nodeType == 0) {
            output.print("X");
        }
        if (maze[i][j].nodeType == 1) {
            output.print(" ");
        }
        if (maze[i][j].nodeType == 2) {
            output.print("S");
        }
        if (maze[i][j].nodeType == 3) {
            output.print("G");
        }
        if (maze[i][j].nodeType == 4) {
            output.print(".");
        }
    }
}
```

Listing 5: *Exclusive ifs* that should be a switch case.

5.2.5 Symptom of a Larger Problem. In several cases, the exclusive `ifs` anti-pattern co-occurs with a deeper structural issue. We reviewed all such examples with the course instructor, who agreed that the co-occurring issues were more important than the anti-pattern. The code in Listing 4 would still be problematic even if it used `if-else`. This code also returns Boolean literals rather than returning a Boolean expression directly (an anti-pattern discussed in [16, 29, 41]). More importantly, the helper method is doing two things instead of one: it determines if two matrices can be added or multiplied, with a parameter indicating the target operation. A cleaner solution would have one method for each action. Listing 6 shows a related problem: lack of a helper method. This code, from the balanced symbol checker, hard-codes closing symbols in a series of `ifs` (while exclusive, they aren't flagged because of the return statements). Instead, a helper method should identify matching symbols. Both listings suggest possible difficulties with abstraction.

```
...
else if (currentChar == '}') {
  if (stack.isEmpty())
    return unmatchedSymbol(rowNum, colNum, currentChar, ' ');
  if (stack.peek() == '(')
    return unmatchedSymbol(rowNum, colNum, currentChar, ')');
  if (stack.peek() == '[')
    return unmatchedSymbol(rowNum, colNum, currentChar, ']');
  stack.pop();
} else if (currentChar == ']') {
  if (stack.isEmpty())
    return unmatchedSymbol(rowNum + 1, colNum, currentChar, ' ');
  if (stack.peek() == '(')
    return unmatchedSymbol(rowNum, colNum, currentChar, ')');
  if (stack.peek() == '{')
    return unmatchedSymbol(rowNum, colNum, currentChar, '}');
  stack.pop();
}
```

Listing 6: Sequential `ifs` co-occuring with the lack of a helper method. This excerpt shows 6 of the 9 `if`-statements that hard-code matched symbols.

Exclusive `ifs` can also co-occur with unnecessary code (Listing 7). Example 1 shows an unnecessary `for` loop, with `if`-statements checking all values of the index; all of those control structures could be removed. We note that usage of `else-if` can also co-occur with this problem, as in Listing 7, Examples 2 and 3. Example 2 includes an `else-if` that can never execute, and Example 3 shows an `else-if` and `else` with identical bodies. These examples caution us against using the presence or absence of the anti-pattern as a measure of overall code quality or conceptual understanding.

```
Example 1: An unnecessary for loop.
for (int i = 0; i < 5; i++) {
  if (i == 0) {
    output.println('{');
    output.println("<a>");
    output.println("a");
    output.println('}');
    output.println();
  }
  if (i ==  1) {
    output.println('{');
    output.println("<b>");
    output.println("b");
    output.println('}');
    output.println();
  ...
}
Example 2: An else-if that will never execute.
if (!lineScan.hasNext()) {
    return unmatchedSymbol(lineCount, colCount,currVal.charAt(j),
      stack.peek().toString().charAt(0));
}
else if (!scan.hasNextLine() && !lineScan.hasNext()) {
    return unmatchedSymbolAtEOF(currVal.charAt(j));
}

Example 3: An else-if block that is redundant with the else block.
if (size == 0) {
    Node<T> newNode = new Node<T>(element);
    head = newNode;
    tail = newNode;
}
else if (size == 1) {
    Node<T> newNode = new Node<T>(element);
    newNode.next = head;
    head.previous = newNode;
}
else {
    Node<T> newNode = new Node<T>(element);
    newNode.next = head;
    head.previous = newNode;
}
```

Listing 7: Unnecessary code co-occurring with exclusive `ifs` (Example 1) and with `else-if` (Examples 2 and 3).

5.3 RQ3: Online Resources Lack Explanation on When to Use `else-if`/`else` vs. Sequential `ifs`

We examined online textbooks from our university library, and free online resources. We chose Java because it is the language used in our university's intermediate programming courses. While our exploration is not an exhaustive review, it covers the resources that our students are likely to find with a quick internet search. We examined six textbooks from our university library: *The Java Programming Language* [13], *Java Programming* [34], *Introduction to Programming Using Java* [18], *Java Programming for Beginners* [30], *Java Programming 24-Hour Trainer* [20], and *Teach Yourself Java in 21 Days* [31]. For the tutorial websites, we did Google search with "java" and each of the following keywords: "conditional structures," "decision structures," and "else-if," checking the top 7 results for each search. This process yielded 11 online resources (many came up in multiple searches): *Java-Decision Making* from turorialspoint [5], *Decision Structures* [4], *Decision Making in Java* by GeeksforGeeks [11], *Decision Making in Java* by DataFlair [7], *Decision Making in Java* by TechVidvan [6], *Java Decision Making* by w3school[2], *Java If-else Statement* [3], *Control Structures in Java* [25], *How to Use If…Else Statements in Java* [22], *Else if statement in Java* [8], and *if/else* by khan academy [9].

We looked specifically for: (1) an explanation that a condition in an `if-else-if`-chain is only evaluated if all previous conditions in the chain have evaluated to false, and (2) guidance on choosing an `else-if` structure when the conditions are exclusive and `else` for the last statement when the conditions are also exhaustive. All of the textbooks address `else`, and all but one ([30]) include `else-if`. (We examined all chapters of [30] that seemed relevant:

branching, data structures and functions). However, none of the textbooks discuss the difference in code execution of `if-else-if` vs. `if-if` structures when the conditions were exclusive, and none offer guidance on when one structure is more appropriate than the other. Even the most in-depth explanation, from *Java Programming* [36], only noted that, for two sequential `if`-statements, "the second if statement will be executed irrespective of whether the first if statement is evaluated to true or not" [36, section 2.2.2].

Of the online resources, many explain the execution of `else-if`, but none compare the execution of exclusive `ifs` with `if-else-if` or `if-else` and none explain when each structure is more appropriate and why. The most comprehensive explanation of the online tutorials, from khan academy, gives examples of code with sequential `ifs` with nonexclusive conditions and compares them with `else-if`, noting how the functionality and execution differ. However, it did not explain the difference in machine execution between `if-if` and `if-else-if` when the conditions *are* exclusive.

6 DISCUSSION

Overall, 30% of the exclusive checks across all assignments were exclusive `ifs`, indicating that this anti-pattern is common. The prevalence of the anti-pattern varied by assignment, with the fewest on #1 and the most on #7. However, we do not interpret raw prevalence as an indication of student knowledge, as the assignments differed in how much they lent themselves to checking exclusive conditions (see section 5.1). Assignment 8, with the highest number of exclusive checks, required creating a binary tree and a priority queue backed by a binary heap, in which students used exclusive checks to decide which side of the tree to traverse. While our data shows which assignments are most likely to *elicit* exclusive checks, it does not show which assignments *should* include exclusive checks in their most elegant implementations. Still, even if we use the percentage of submissions with exclusive checks as a proxy for opportunities to use them, our data does not show improvement across the semester. Considering the assignments where at least 10% of submissions used exclusive checks (all but #1), the 3 highest rates of using exclusive `ifs` (out of the total number of submissions with exclusive checks) occur in each third of the semester, on assignments 2, 7, and 12. Thus, consistent with prior work [17, 27], we cannot conclude that the propensity to use the anti-pattern declines over CS2. This is not surprising given that students were not incentivized to avoid exclusive `ifs` anti-pattern.

Although many submissions included the anti-pattern, many *also* included `if-else-ifs`. This suggests that a student's usage of an anti-pattern may not mean that the student cannot use the correct pattern. We found that 142 students used both the pattern and anti-pattern in the same submission. Another 18 students used the pattern in one submission and then used the anti-pattern in a later assignment. These 160 students are 59% of the class. While our field has mainly focused on detecting anti-patterns [16, 17, 27, 38], this may provide an incomplete picture of students' knowledge of code structure. If pattern detectors seek out correct usages in addition to anti-patterns, when they flag those anti-patterns, they could remind students of where a correct example is *in the students' own code*. Further, if students are capable of using the correct pattern, instruction on that pattern may not be as effective as instruction on identifying opportunities for using the pattern.

More important than the anti-pattern itself, we found larger co-occurring problems. This finding shows promise for exploring anti-patterns as symptoms of more important issues that are harder to detect directly. Conversely, it cautions against providing feedback to students that may help them revise the anti-pattern without considering the deeper issues in their code.

6.1 Threats to Validity

Our human ground truth is limited by the difficulty of determining side effects through hand-inspection. Also, our data shows what students did but not why they did it – use of the exclusive `ifs` anti-pattern may be due to flaws in logical reasoning. Our data includes noise inherent to the ecological context (e.g., some students dropped the class, started late, or did not submit all assignments). And, since students worked in pairs, instances of using the pattern and anti-pattern on the same assignment could reflect the work of different students. Finally, while this anti-pattern is explored in prior work and is important to our instructors, this study does not examine its importance in professional settings.

6.2 Implications for Educators

Since `else-if` is taught in CS1, CS2 instructors may assume that students know how to use it properly. However, our exploration of 17 online resources suggests an instructional gap on this structure. Our CS2 instructor is considering using our data as a wake-up call to students in his class, to show them that while they think they are not susceptible to this anti-pattern, it is common and they should look out for it. Instruction on this anti-pattern should provide examples from each of the sub-categories, should focus on identifying opportunities to revise the anti-pattern, and should emphasize the importance of looking for deeper structural issues that may co-occur. Instruction should also draw on students' prior knowledge of the correct pattern, since students are likely capable of both. Consideration of these deeper issues is also important in courses where students are given automated feedback on structure.

7 CONCLUSION AND FUTURE WORK

We developed a detector for the exclusive `ifs` anti-pattern by combining static analysis with an SMT solver. Running the detector on students' programming assignments showed this anti-pattern was common in CS2. Hand inspection revealed sub-categories of the anti-pattern. Online resources lacked instruction on this anti-pattern, making it unlikely that students will learn it on their own.

In future work, we plan to refine the detector and examine how instructors might use it in their courses. We advocate think-aloud studies and interviews to examine why students use anti-patterns, further explorations of how usage of anti-patterns relates to conceptual knowledge, and the development of additional detectors for complex anti-patterns.

8 AKNOWLEDGEMENTS

We thank Dr. Daniel Kopta, Dr. David Johnson, and Dr. Anna Rafferty for their guidance and feedback. This work was supported by the National Science Foundation through award SHF 1948519.

REFERENCES

[1] 2016. https://clang-analyzer.llvm.org/

[2] 2019. Java decision making. https://www.w3schools.in/java-tutorial/decision-making/

[3] 2019. Java if else - javatpoint. https://www.javatpoint.com/java-if-else

[4] 2020. Decision Structures. https://www2.lawrence.edu/fast/GREGGJ/CMSC150/012Decisions/Decisions.html

[5] 2020. tutorialspoint. https://www.tutorialspoint.com/java/java_decision_making.htm

[6] 2021. Decision making in java - explore the types of statements with syntax. https://techvidvan.com/tutorials/decision-making-in-java/

[7] 2021. Decision making in java (syntax amp; example)- A complete guide for you! https://data-flair.training/blogs/decision-making-in-java/

[8] 2021. Else if statement java. https://code-knowledge.com/java-elseif/

[9] 2021. If/else - part 2 | logic and if statements | Intro to JS: Drawing amp; animation | computer programming | computing. https://www.khanacademy.org/computing/computer-programming/programming/logic-if-statements/pt/ifelse-part-2

[10] 2022. Checkstyle 10.3.2. https://checkstyle.sourceforge.io/index.html

[11] 2022. Decision making in Java (if, if-else, switch, break, continue, jump). https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/

[12] 2022. pmd source code analyzer. https://pmd.github.io/latest/

[13] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java programming language*. Addison Wesley Professional.

[14] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[15] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference*. 73–82.

[16] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference*. 73–82.

[17] Tomche Delev and Dejan Gjorgjevikj. 2017. Static analysis of source code written by novice programmers. In *2017 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 825–830.

[18] David J Eck. 2015. *Introduction to programming using Java*. David J. Eck.

[19] Tomáš Effenberger and Radek Pelánek. 2022. Code Quality Defects across Introductory Programming Topics. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 941–947.

[20] Yakov Fain. 2015. *Java Programming 24-Hour Trainer*. (2nd ed.. ed.).

[21] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[22] James Gallagher. 2020. How to use if...else statements in Java. https://careerkarma.com/blog/java-if-else/

[23] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating Pedagogical Code Reviews into a CS 1 Course: An Empirical Study. *Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE'09)* (2009), 291–295.

[24] Saj-Nicole A Joni and Elliot Soloway. 1986. But My Program Runs! Discourse Rules for Novice Programmers. *Journal of Educational Computing Research* 2, 1 (1986), 95–125.

[25] Nickson Joram. 2021. Control structures in Java-conditional statements. https://medium.com/javarevisited/control-structures-in-java-conditional-statements-e4d8da0421cc

[26] Oscar Karnalim, William Chivers, et al. 2022. Work-In-Progress: Code Quality Issues of Computing Undergraduates. In *2022 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 1734–1736.

[27] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 110–115.

[28] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 119–125.

[29] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 562–568.

[30] Mark Lassoff. 2017. *Java programming for beginners* (1st ed. ed.). PACKT Publishing.

[31] Laura Lemay, Charles L Perkins, and Michael Morrison. 1999. *Teach yourself Java in 21 Days*. Sama Publishing.

[32] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[33] Andy Oram and Greg Wilson. 2007. *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. O'Reilly Media, Inc.

[34] Hari Pandey. 2011. *Java Programming* (1st edition. ed.).

[35] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 410–415.

[36] K Rajkumar. 2013. *JAVA Programming* (1st edition. ed.).

[37] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Trans. Software Eng.* SE-10, 5 (1984), 595–609.

[38] Leo C Ureel II and Charles Wallace. 2019. Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 738–744.

[39] Nathaniel Weinman, Armando Fox, and Marti A Hearst. 2021. Improving instruction of programming patterns with faded parsons problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–4.

[40] JL Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient elements in novice solutions to code writing problems. (2011).

[41] Eliane S Wiese, Anna N Rafferty, and Armando Fox. 2019. 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, 84–94.

[42] Eliane S Wiese, Anna N Rafferty, and Garrett Moseke. 2021. Students' Misunderstanding of the Order of Evaluation in Conjoined Conditions. In *ICPC'21: Proceedings of the 29th International Conference on Program Comprehension*.

[43] Eliane S Wiese, Michael Yen, Antares Chen, Lucas A Santos, and Armando Fox. 2017. Teaching students to recognize and implement good coding style. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 41–50.

[44] M Woodley and S N Kamin. 2007. Programming studio: A course for improving programming skills in undergraduates. *SIGCSE 2007: 38th SIGCSE Technical Symposium on Computer Science Education* January 2007 (2007), 531–535.