Augmenting Code Pattern Detection with Software Verification and Examining How Teaching Assistants Interact with Student Code Structure

Matthew Hooper
University of Utah

UUCS-21-014

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

10 May 2021

Abstract

Expert programmers expect certain commonly-used patterns in code. However, novice coding patterns often deviate from these expectations, even if they're still functionally correct. University instructors recognize the need to educate their students to use more advanced patterns, but giving personalized feedback is time-consuming, and teaching good coding style and structure doesn't have an obvious home in a 4-year curriculum. It may be possible to employ software that can automatically identify novice coding patterns, or even to instruct students about their style without using the limited time and resources of course instructors. Since teaching assistants are viewing and giving feedback about student code more often than instructors, it may be possible to use their experiences to design effective feedback for an automated pattern detector. How teaching assistants interact with students specifically about their code structure appears not to be studied previously. Through several interviews with teaching assistants I've found that teaching assistants don't always agree with expert coding patterns, and their reasons for whether or not they agree may be vague or poorly articulated - demonstrating a lack of mastery over certain programming topics. In addition, by applying techniques from software verification, I've been able to expand the kinds of novice patterns that can be detected by existing pattern detection tools.

AUGMENTING CODE PATTERN DETECTION WITH SOFTWARE VERIFICATION AND EXAMINING HOW TEACHING ASSISTANTS INTERACT WITH STUDENT CODE STRUCTURE

by

Matthew Hooper

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the Degree

Bachelor of Computer Science

School of Computing
The University of Utah
April 2021

Approved:

Eliane Wiese /
Eliane Wiese, PhD
Thesis Faculty Supervisor

/
H. James de St. Germain
Director of Undergraduate Studies
School of Computing

/
Director, School of Computing

Acknowledgements

To my parents Bob and Nancy, who always knew I'd one day amount to something, even when I didn't.

To my brother James, for suggesting I go to school and check out this programming thing, and making this suggestion at least once a year, for nearly a decade.

To Joe and Josh, for separately telling me that the world is my oyster, if only I'd get over my aversion to seafood.

To Serena, for finding my comma splices only for me to splice some new ones.

To Astro and Nebula, whose walks were always a welcome break from my desk.

To Dr. Wiese, for whom this never would have happened without her particular brand of encouragement.

Abstract

Expert programmers expect certain commonly-used patterns in code. However, novice coding patterns often deviate from these expectations, even if they're still functionally correct. University instructors recognize the need to educate their students to use more advanced patterns, but giving personalized feedback is time-consuming, and teaching good coding style and structure doesn't have an obvious home in a 4-year curriculum. It may be possible to employ software that can automatically identify novice coding patterns, or even to instruct students about their style without using the limited time and resources of course instructors. Since teaching assistants are viewing and giving feedback about student code more often than instructors, it may be possible to use their experiences to design effective feedback for an automated pattern detector. How teaching assistants interact with students specifically about their code structure appears not to be studied previously. Through several interviews with teaching assistants I've found that teaching assistants don't always agree with expert coding patterns, and their reasons for whether or not they agree may be vague or poorly articulated - demonstrating a lack of mastery over certain programming topics. In addition, by applying techniques from software verification, I've been able to expand the kinds of novice patterns that can be detected by existing pattern detection tools.

Contents

List of Figures

1 Introduction

The topic of research is how novice structure [17] affects the ability of course staff (professors, teaching assistants) to provide assistance with student code. Novice coding structures are patterns of code that are easy for novice programmers to write to achieve a correct function, but are different from what a more experienced programmer would expect to see to accomplish the same function. Previous research shows us that novice code structure is prevalent, it's undesirable, and learning how to teach better structure is worth investigating [17]. Novice code structure is not just present in the first year courses, some of these patterns are still prevalent into students' second and third years [17].

University instructors are interested in addressing novice structure issues but may lack class-time resources to teach, and may not realize the full extent of usage of novice patterns among their students; this is in part because instructors don't see lots of student code [1]. Instead, its usually the teaching assistants that see the most student code. While there is research about the use and education of style in novice coders [6, 1], there does not appear to be research on how teaching assistants help students with novice structure. While there is some research about the kinds of help students receive during TA office hours [13], existing research focuses on development and correctness issues, rather than coding structure.

Having worked for the School of Computing as a teaching assistant for several years, myself and other TA's have experiences demonstrating the impact novice structure can have on effective instruction. One colleague shared with me an instance where poor coding structure among students was endemic enough to change course policy [3]. In a 4000-level course TA's were spending so much more time trying to understand poorly structured code than they were on identifying actual bugs, that it caused huge delays in students' queue time waiting for a TA. This eventually caused a change in policy for this course: If students brought poorly structured code to office hours, they would be ineligible for TA assistance until it was improved. After this policy was put in place, the TA's noticed a marked improvement in the quality of code that their students brought to them. This suggests students are in fact capable of using good structure in their code, however they may not yet be proficient with using expert structure, or require incentive or motivation to do so.

I contributed to the development of an automated tool to detect the novice code structures that we're interested in. PMD [11] is an open-source static analysis tool for Java and other programming languages. Developing an extension of PMD to target novice code structures has been an ongoing development effort under Dr. Eliane Wiese's research. This project includes development of a detector for the 'exclusive ifs' pattern using techniques other than the usual AST analysis. The detector was used to identify code samples for use in the teaching assistant interviews.

Second, I conducted a study with School of Computing teaching assistants and interviewed them about their thoughts and opinions with novice code, and how it relates to their experiences assisting their students. They were shown samples of code that exhibited various forms of novice style. I asked them for their evaluation of the sample, and what kinds of instruction they'd give if the sample came from one of their students. This study was approved by the IRB for Dr. Wiese, with additional amendments completed by myself for recruiting TA's. The interviews were analyzed using thematic analysis [2], a qualitative analysis technique, to identify common ideas and experiences among the interviewees.

2 Background and Related Work

Novice coding structures are patterns of code that are easy for novice programmers to write to achieve correct function, but are different from what a more experienced programmer would use. Novice structure may be exhibited as poor use of abstraction, such as classes, structs, and methods, failing to use language features and syntax in an idiomatic manner, or naïve patterns that could be more succinct or simplified. Experts expect a certain structure to code, and code that does not conform to these expectations can be more challenging to read and understand [14]. However, novices don't necessarily perceive the expert structures as more readable. [17]

The kinds of novice patters that we're interested in studying are discussed by Wiese et al. [17]. Three patterns of particular interest are shown here. Figure 1 shows the Boolean Return pattern, where a condition is tested and a Boolean method returns a literal `true` or `false`, rather than returning the condition itself. In the expert example, a more experienced programmer would understand that the `condition` expression is being evaluated and returned, whereas a novice programmer might prefer to have the logic "spelled-out" with the `if-else` statement.

Figure 2 shows an example of a `while` loop that should be a `for` loop, since the conventional wisdom is that a `for` loop should be used when the number of iterations is always known at runtime [7, 16].

Finally, Figures 3 and 4 demonstrate the "exclusive ifs" pattern; a series of consecutive `if` statements that should be using `else if` and `else` statements instead. An in-depth discussion of this pattern follows in section 3.1.

```
boolean noviceBooleanReturn() {
    boolean condition;

    // some work
    if (condition) {
        return true;
    }
    else {
        return false;
    }
}
```

```
boolean expertBooleanReturn() {
    boolean condition;

    // some work

    return condition;
}
```

Figure 1: Novice and Expert examples of the Boolean Return pattern.

```
void noviceLoop(int n) {
    int count = 1;                void expertLoop(int n) {
    while (count < n) {               for (count i = 0; count < n; count++) {
        // some work;                     // some work;
        count++;                      }
    }                             }
}
```

Figure 2: Novice and Expert examples of the loop pattern.

```
String noviceExclusiveIfs(String s) {
    if (s.length < 10) {
        return "small";
    }
    if (s.length >= 10 && s.length < 20) {
        return "medium";
    }
    if (s.length > 20) {
        return "large";
    }
    return "";
}
```

Figure 3: Novice example of the Exclusive Ifs pattern.

```
String expertExclusiveIfs(String s) {
    if (s.length < 10) {
        return "small";
    }
    else if (s.length < 20) {
        return "medium";
    }
    else {
        return "large";
    }
}
```

Figure 4: Expert example of the Exclusive Ifs pattern.

Previous research shows that University instructors recognize the need to instruct better code structure [1]. However, professors don't typically see many examples of student code, and manually reviewing samples would be incredibly time consuming. As a result instructors are admittedly unaware of how common the use of novice structure is, and typically underestimate its prevalence. Instructors would like to instruct their students to use better structure, but lack the time and space in their curriculum to give meaningful instruction [1].

Many instructors at the University of Utah suggest CS 1410 as the appropriate starting point for teaching good structure [1]. Instructors that teach this course find that teaching good structure here can be challenging since this is a beginner course. Students are typically more concerned with getting their program to compile and execute. Telling a student at this level that their code works, but wasn't written *correctly* can be confusing, frustrating, and demoralizing [1].

It appears to be the case that teaching students to use expert structure is important, but there isn't an obvious place for it in the curriculum. In beginning courses, students may not be ready to think about their code at that level, and in later courses, good code structure may be considered outside the scope of the learning goals.

We may be able to employ a tool that can automatically detect novice patterns to address these issues. Such a tool could be used on batches of student assignment for quantitative analysis of the use of novice code by students. It would also enable instructors to quickly identify and give appropriate feedback to their students. We may also be able to use it to assist the instruction of expert style as well. A tool integrated into the student's IDE's, such as a linter, could also give feedback to students about their structure while they're writing their code. If students are able to engage with the linter, and the feedback students receive is effective, then they could receive instruction about more advanced code structures without requiring instructors to dedicate limited time and resources.

Existing linter and static analysis tools appear to not fit our needs for several reasons. First, most of the patterns they detect are not related to novice style. For example, SonarLint [15], PMD [11], and Roslynator [10] offer static analysis and linting with hundreds of rules or patterns in C# or Java. Only a handful of rules actually detect the novice patterns we're interested in. The rest are focused on fixing bugs, security vulnerabilities, or adherence to naming, whitespace, and bracketing conventions.

In addition, existing tools do not detect many of the novice style patterns that we're interested in. While some patterns such as collapsible consecutive or nested if statements are commonly included in existing tools, others are only intermittently included, such as detecting while loops that should be for loops. Other patterns aren't included at all and may be difficult or impossible to detect with static analysis techniques. For example, the 'exclusive ifs' pattern could only be detected with AST analysis in simple cases. Using other techniques and tools such as an SMT solver or concolic execution could greatly increase the number of instances that would be detected. The efforts applied in detecting this particular pattern are described in detail.

Lastly, the feedback given may not be appropriate for an educational setting. The feedback is usually given in the context of alerting an expert of an issue, assuming they already know how to fix it. Feedback given to a novice needs to be given in such a way that they can engage with it. Feedback with too technical of language might be dismissed or ignored.

A pattern detection tool would also need to be designed to support course staff as well. In my personal experience as a TA, during one semester we had an autograding tool that students could test their homework against before making their final submissions. Frequently, the autograder would produce strange results or even results that were incorrect. This autograding tool was maintained by a lab instructor, and this individual was not always accessible to the students to address their questions about their autograder results. This meant that students would go to the TAs with their questions. Since the tool was not really accessible to the TAs either, the TAs would often instruct students to dismiss or ignore the autograder altogether, undermining its usefulness.

One can imagine a similar scenario playing out with a pattern detector. If course staff often disagree with, or don't understand the feedback given, they may instruct students to discount the advice they receive, defeating the purpose of using a pattern detector. Since teaching assistants are still students, we can't assume their level of expertise with regards to code structure.

This leads us to consider the following research questions:

- Are teaching assistants more likely to agree with experts or novices with regards to code structure?
- How do teaching assistants communicate with students about code structure? For instance, they may notice and give feedback about structure if it makes it difficult for them to debug. Alternatively, they may not give feedback about code structure if they don't think it's their role to do so, or for other reasons.
- Can we use software to identify the novice structures that we're interested in?
- Since the course TA's are on the front lines with the students, looking

at their code and providing feedback, can we use their experience to inform the design of a pattern detector so it gives informative, actionable feedback?

3 Pattern Detector

3.1 Methods

An extension to PMD, an open-source source code analyzer, has been under ongoing development through Dr. Wiese's research as a tool to automatically detect instances of novice structure patterns [1, 12]. One pattern that was yet undetected by PMD is the 'exclusive ifs' pattern. This pattern is simply when students use an `if` statement where an `else if` statement would be more appropriate, as demonstrated in Figure 3.

Since checking for `i < 10` is covered by the first `if` statement, the second `if` statement can be changed to an `else if` statement, and we can omit the redundant `i >= 10` check. In addition, since since these three `if` statements account for all possible values of `i`, we can replace the final `if` statement with an `else`. Once we've made these changes, we can remove the unreachable `return ""` at the end of the method, since the compiler will find that all execution paths have a `return` value. The result of applying these changes can be seen in Figure 4.

These changes are preferable since it makes the code easier to read, easier to maintain, and less likely to include errors. First, the use of `else` and `else if` communicate to the reader that these conditions are meant to be exclusive, and they enforce this exclusivity at runtime. In the original program, it's not clear why there's a hanging `return ""` statement. Is this supposed to represent some error state? Is there a case that should be handled by the code, but was missed by the developer? Eliminating this `return` statement communicates that these 3 cases are meant to handle all possible inputs. Second, eliminating the redundant `i >= 10` condition from the second `if` statement makes the code easier to maintain. If the bounds on what was considered "small" and "medium" changed, the expert version only needs to be updated in a single location, whereas the novice version requires updating 2 lines of code.

Instances of this pattern would be difficult to detect with the analysis that the pattern detector currently uses. Currently, PMD analyzes source code by examining the Abstract Syntax Tree of the program. For example, PMD can detect the Boolean return pattern by looking for an AST with a particular structure (Figure 5).

However, when checking for the exclusive ifs pattern, simply identifying the AST structure is not sufficient. A pair of `if` statements that are exclusive (Figure 6) could have the same AST structure as a pair that are not exclusive

```
boolean noviceReturn() {
    boolean condition;

    // some work

    if (condition) {
        return true;
    }
    else {
        return false;
    }
}
```

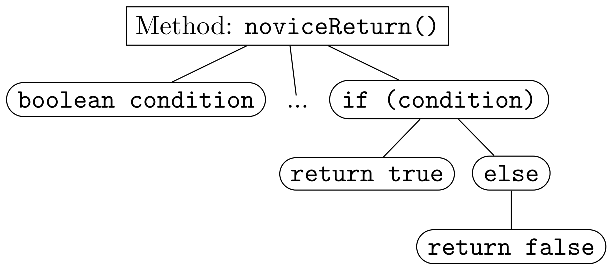


Figure 5: AST for the novice Boolean return pattern. Detecting the subtree starting at `if (condition)` is sufficient to identify the pattern.

(Figure 7). This means that once the structure is identified, it's necessary to test the `if` statements for exclusivity. PMD can examine the conditions of the `if` statements, and can test things such as the type of expression, or notice if the conditions reference the same variables. However it doesn't include a way to test for exclusivity. It would be possible to develop a program to do just that, starting with simple equality or arithmetic checks, and then developing from there. The logical conclusion of extending this program would be a SAT solver, since we want to test if the conjunction of two Boolean expressions is satisfiable.

```
void notExclusive(int i) {
    if (i <= 0) {
        foo();
    }
    if (i >= 0) {
        bar();
    }
}
```

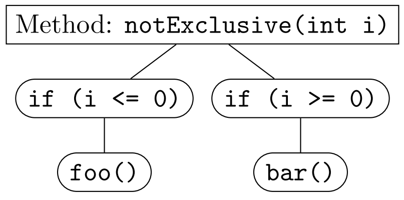


Figure 6: Non-exclusive if statements with AST.

Furthermore, side-effects of function calls could make the analysis difficult. Consider the example program in Figure 7. Suppose `i` was a global variable rather than a function parameter. This means the `foo()` method could modify the value of `i`, and as a result this pair of statements wouldn't actually be exclusive.

This means that PMD can be used to detect AST structures of potentially

```
void exclusive(int i) {
    if (i == 0) {
        foo();
    }
    if (i != 0) {
        bar();
    }
}
```

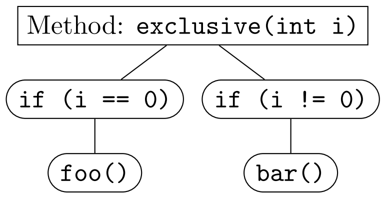


Figure 7: Exclusive if statements with AST.

exclusive `if` statements, but actually testing the statements for exclusivity requires more specialized programs.

Lastly, developer intent is a black box. It is not possible to examine the structures in a program and determine if its author used those structures intentionally or accidentally. In particular, it may always be the case that the author purposely did not use an `else if` statement when it would appear to be preferable. Even so, it would be useful to point such instances out, and let the developer decide what to use.

In collaboration with Dr. Zvonomir Rakamaric, we employed techniques from software verification to detect this pattern and address these issues. We developed two approaches, one with a dynamic analysis technique, and one analyzing the code statically. Ultimately, we incorporated the static approach into the pattern detector, though further efforts in employing the dynamic approach are worth investigating.

3.1.1 Identifying `if` statements to test

We define a set of consecutive `if` statements to be exclusive if only one of the `if` statements ever executes when that part of the program is reached. In such a set, we only need to test the consecutive pairs of `if` statements for exclusivity.

```
if (condition1) {...}
if (condition2) {...}
if (condition3) {...}
```

Figure 8: Consecutive if statements. The `if` statements for `condition1` and `condition2` form a consecutive pair, as do `condition2` and `condition3`.

Take Figure 8 for example, if we find that the `if` statements for `condition1` and `condition3` are exclusive, it would be incorrect to suggest changing `condition3` to use an `else if` since this would make `condition2` and `condition3` exclusive. In some cases it may be possible to rearrange the `if` statements so that `condition3` could use an `else if`, but it's not always possible to do so in a way that preserves semantic equivalence of the program. Detecting such cases remains an interesting challenge not explored by this project.

Once identified, testing the pairs of `if` statements for exclusivity varies under each approach. In the dynamic approach, we use concolic execution to test if both `if` statements are ever executed. Under the static approach we take the conditions of each `if` statement and test their conjunction for satisfiability.

A simple approach would be to check `if` statements that appear as siblings in the AST. Consider the example in Figure 9:

```
void example() {
    if (condition1) {
        foo();
    }
    if (condition2) {
        bar();
    }
}
```

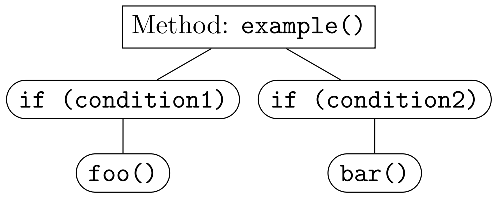


Figure 9: A simple example with AST.

The `if` statements are represented as child nodes of the method `example` node. The bodies of the `if` statements are represented as their children. In this case, the `if` statements for `condition1` and `condition2` are siblings in the AST. In addition, we also want to consider the example in Figure 10 for transformation:

```
void endingElse() {
    if (condition1) {
        foo();
    }
    if (condition2) {
        bar();
    }
    else {
        baz();
    }
}
```

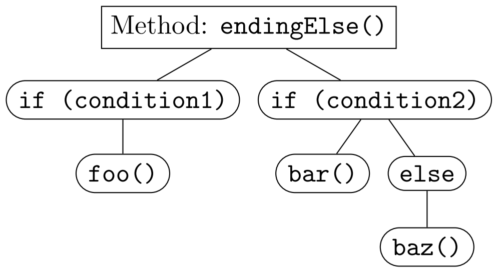


Figure 10: Even though `condition2` has an `else` statement, we still want to test for exclusivity with `condition1`.

The first two `if` statements are consecutive in the source code. If `condition1` and `condition2` are exclusive, it would be more appropriate for `condition2` to use an `else if`.

However, it turns out this simple approach is neither a necessary nor sufficient condition. There are some patterns we should test for exclusivity, even when the `if` statements are not siblings in the AST, and there are some sibling `if` statements that we should not test. The code snippet in Figure 11 is an example of both cases.

```
void ifElseIfIf() {
    if (condition1) {
        foo();
    }
    else if (condition2) {
        bar();
    }
    if (condition3) {
        baz();
    }
}
```

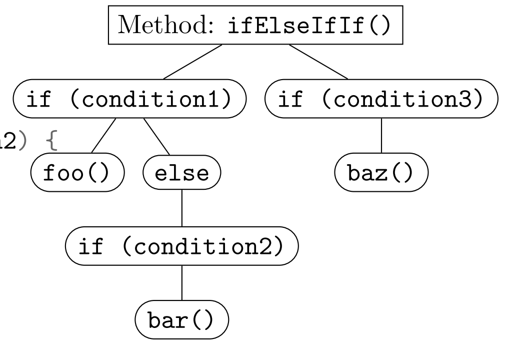


Figure 11: ‘Siblings in the AST’ is not the same as ‘Consecutive in the source code.’

The `if` statements for `condition2` and `condition3` are consecutive in the source code, so we want to check if they are exclusive. In addition, we don’t want to test `condition1` and `condition3` for exclusivity, since changing `condition3` to use an `else if` would make it exclusive with `condition2`, not `condition1`.

Notably, even though the `if` statements for `condition2` and `condition3` appear consecutive in the source code, they are not adjacent in the AST. Instead, `condition2` is a descendent of `condition1`. Since there can be any number of `else if` descendants, such as in Figure 12, it’s necessary to check for the deepest `if` descendant. In Figure 12, only `conditionN-1` and `conditionN` should be checked for exclusivity. We do not want to check `condition1` and `conditionN`.

```
void nElseIfs() {
    if (condition1) {
        foo();
    }
    else if (condition2) {
        bar();
    }
    ...
    else if (conditionN-1) {
        baz();
    }
    if (conditionN) {
        qux();
    }
}
```

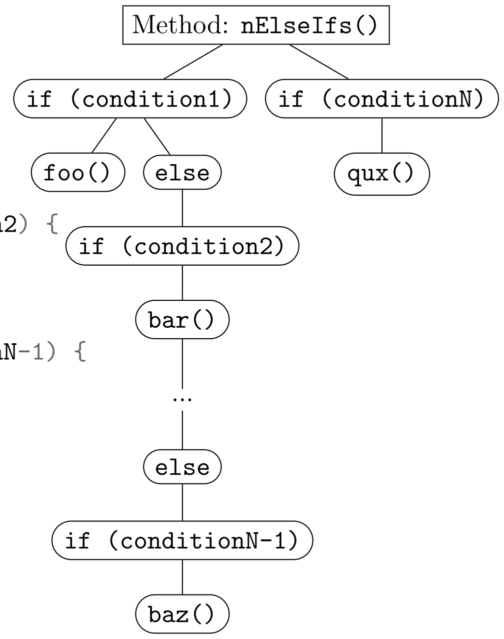


Figure 12: There can be any number of `else if` statements between `condition2` and `conditionN-1`.

The code in Figures 13 and 14 are examples of `if` statements that are not consecutive in the source code, and therefore are not tested for exclusivity:

```
void notConsecutive() {
    if (condition1) {
        foo();
    }
    bar();
    if (condition2) {
        baz();
    }
}
```

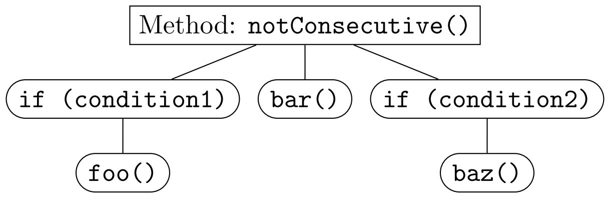


Figure 13: Any statement between two `if` statements means they aren't consecutive.

Note that in Figure 14, the `if` statements for `condition1` and `condition2` do appear as siblings. However, in both examples it would be a syntax error to change the `if` statement for `condition2` to an `else if` statement, so we do not want to test these conditions for exclusivity.

Not only do we need to identify `if` statements in these patterns, we also

```
void ifElseIf() {
    if (condition1) {
        foo();
    }
    else {
        bar();
    }
    if (condition2) {
        baz();
    }
}
```

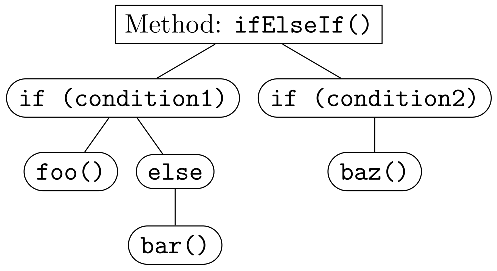


Figure 14: The `else` statement comes between the two `if` statements.

need to consider the body of the `if` statements. In Java, there are 4 branching control flow statements which modify the order of execution of a program. They are `return`, `break`, `continue`, and `throw`. These statements impose a sort of exclusivity already; if inside the body of an `if` statement, they would prevent the execution of a following `if` statement, just as an `else if` statement would. This means the programs in Figure 15 have identical execution:

```
void branchingControlFlow1() {
    if (condition1) {
        foo();
        return;
    }
    if (condition2) {
        bar();
    }
    baz();
}
```

```
void branchingControlFlow2() {
    if (condition1) {
        foo();
        return;
    }
    else if (condition2) {
        bar();
    }
    baz();
}
```

Figure 15: Since `condition1` will always `return` from the method, using an `else` statement for `condition2` doesn't change how this program will execute.

The `else if` is unncessary, since if `condition1` is true, then `condition2` is not executed in either example. Since neither structure is preferable, we don't want our pattern detector to recommend changing the `if` statement for `condition1` to an `else if`. Note that this is another example of two adjacent `if` statements in the AST that we wish to ignore, as see in in Figure 16. Finally, its necessary to apply all of the above logic at each level of scope in the source code.

In Figure 17, `condition1`and `condition4` should be checked for exclusivity,

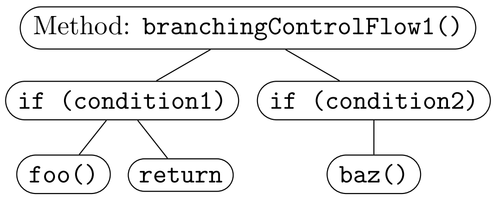


Figure 16: AST for the `branchingControlFlow` method.

```
void scopeLevels() {
    if (condition1) {
        if (condition2) {
            foo();
        }
        if (condition3) {
            bar();
        }
    }
    if (condition4) {
        baz();
    }
}
```

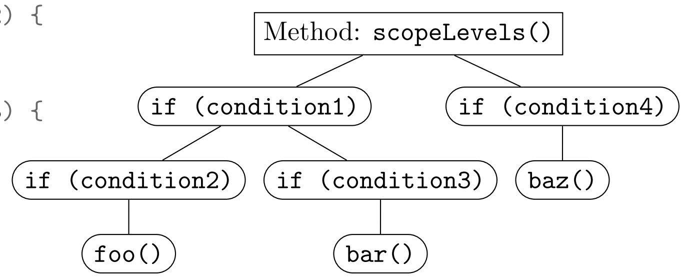


Figure 17: The `if` statements for `condition1` and `condition4` are consecutive in the source code, since `condition2` and `condition3` lie in a child scope.

as well as `condition2` and `condition3`. Every kind of code block should be checked; scopes created by loop statements, method calls, floating scope brackets, etc. Note that this per-scope approach conveniently handles cases where branching control flow statements are conditionally contained inside an if statement, since the conditions will reside in a child scope and can therefore be ignored in this scope level. The child condition will be tested in its own scope level. An example of this is shown in Figure 18. Since `condition1` doesn't necessarily return from the method, its `return` statement doesn't impose any exclusivity with `condition3`. Therefore, we should still check `condition1` and `condition3`.

```
void conditionalReturn() {
    if (condition1) {
        foo();
        if (condition2) {
            return;
        }
    }
    if (condition3) {
        bar();
    }
}
```

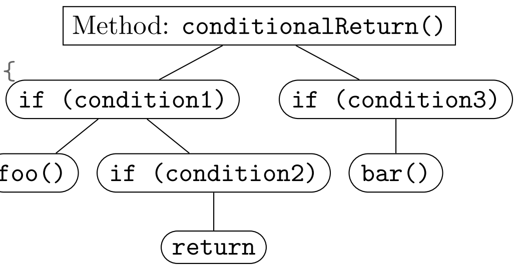


Figure 18: It's still necessary to test `condition1` and `condition3` for exclusivity, since `condition1` only returns from the method under `condition2`.

A general algorithm to identify `if` statements for exclusivity testing is described in Figure 19. Once we've identified the `if` statements to test for exclusivity, how we perform that test varies under the two approaches we developed.

3.1.2 The Dynamic Approach

Dr. Rakamaric has previously worked on a tool called JDart which can perform concolic execution on Java code. [8] Concolic execution is a dynamic analysis technique. Essentially, it tries to run a program with every possible input that completes a different execution path through the program. For example, the output of JDart on a simple program is shown in Figure 20.

It reports the total number of unique execution paths through the program. It also reports the number of paths that threw an exception vs threw no exceptions, and paths for which it is unable to determine a result. While this output does not give the specific values that result in each path, this case is simple enough to deduce. There are 2 "OK" paths: when `i < 1000`, and when `i > 1000`. There is 1 "error" path, when `i == 1000`.

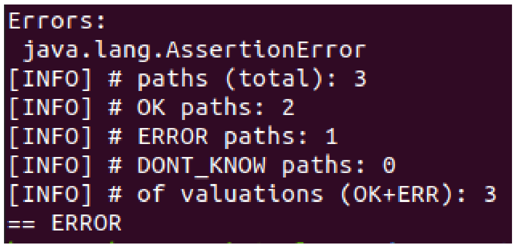
Encoding the Exclusivity Test Under a dynamic view of exclusivity, we consider a pair of `if` statements to be exclusive if for all possible inputs, it is never the case that both `if` statements execute in a given run of the program. Since this test will include running the program, it's necessary to modify the program to encode this definition of exclusivity into our test. We introduce variables to track the execution of each consecutive `if` statement, and a statement to `assert` exclusivity.

```
For each code block c:
    previous = none
    For each statement s in c:
        If s is not an if statement:
            previous = none
            continue
        Else if previous:
            test(s, previous)

        If s terminates with a branching control flow statement:
            previous = none
        Else if s has a child else statement:
            previous = deepest else if child of s
        Else:
            previous = s
```

Figure 19: Algorithm to determine which pairs of `if` statements to test for exclusivity.

```java
void simpleConcolic(int i) {
    if (i >= 1000) {
        if (!(i > 1000)) {
            assert false;
        }
    }
}
```



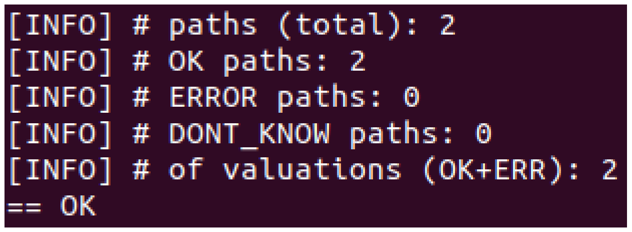
```
Errors:
 java.lang.AssertionError
[INFO] # paths (total): 3
[INFO] # OK paths: 2
[INFO] # ERROR paths: 1
[INFO] # DONT_KNOW paths: 0
[INFO] # of valuations (OK+ERR): 3
== ERROR
```

Figure 20: One execution path reaches the `assert false` statement.

Our initial encoding was as follows: For each pair of `if` statements we wish to test, modify the source code and introduce a new `counter` variable. In each `if` statement, add a new statement to increment the `counter` variable. Finally, add a statement after the pair of `if` statements to assert `counter < 2`. Then run JDart on the program. If JDart finds that all execution paths pass the assertion, then this means both `if` statements were never both executed. Consider the example in Figue 21. Since JDart finds no path that fails to `assert counter < 2`, as seen in Figure 22, it must mean that these `if` statements are exclusive.

```
public void example(int i) {
    if (i <= 10) {
        foo();
    }
    if (i > 10) {
        bar();
    }
}
public void exampleTransformed(int i) {
    int counter = 0;
    if (i <= 10) {
        counter++;
        foo();
    }
    if (i > 10) {
        counter++;
        bar();
    }
    assert counter < 2;
}
```

Figure 21: A simple program before and after transformation for the test.



```
[INFO] # paths (total): 2
[INFO] # OK paths: 2
[INFO] # ERROR paths: 0
[INFO] # DONT_KNOW paths: 0
[INFO] # of valuations (OK+ERR): 2
== OK
```

Figure 22: JDart output from the `exampleTransformed` program.

While this encoding is straightforward for a pair of `if` statements, it's not clear how this approach will scale if there are many consecutive `if` statements. Consider the more complicated program in Figure 23. Since we only check the consecutive pairs, then it's no longer correct to `assert counter < 2`. This would let us know if all the `if` statements should be `else if` statements, but we'd like to test each pair individually. This would require each pair to have its own `counter` variable. Since a single `if` statement could be part of multiple pairings, this means each `if` statement would need to increment each of the `counter` variables for each pair it is part of. The results of applying this transformation are seen in Figure 24.

```
void example2(int i) {
    int x = 0;
    if (i % 2 == 0) {
        x++;
    }
    if (i < x) {
        x = 2;
    }
    if (i > x) {
        x--;
    }
    if (i % 2 == 1) {
        x *= 2;
    }
    return x;
}
```

Figure 23: How to transform a program with more than two consecutive `if` statements?

Instead, we can simplify the above by testing if each branch was executed, rather than counting the number of branches executed per pair. We can do this by using a Boolean variable for each branch to indicate that branch was visited, e.g., `branch1`, `branch2`, `branchN` for N `if` statements. Then for each consecutive pair of `if` statements `branchJ` and `branchK`, we assert `branchJ XOR branchK`. This simpler encoding is seen in Figure 25.

However, this version of the encoding is insufficient for two reasons. First, if two statements were never reached, then the XOR assertion would still be true, but not because the statements were exclusive. Rather, it's because these statements are dead code. While it might be useful to report such instances, this encoding makes dead code indistinguishable from exclusive statements.

To address this issue, we arrived at the final encoding, which involves up to 2 calls to JDart but with different assertions. The first is seen in Figure 26.

Here we can see this assertion will only be true if one or zero branches were executed. If the transformed code always passes the assertion, then either these `if` statements are exclusive, or they are dead code. To determine which is the case, we apply a second transformation (Figure 27) to the original code, and run JDart again. If this assertion always passes, then we've found dead code.

```
void example2Transformed1(int i) {
    int x = 0;
    int pair1 = 0;
    int pair2 = 0;
    int pair3 = 0;
    if (i % 2 == 0) {
        x++;
        pair1++;
    }
    if (i < x) {
        x = 2;
        pair1++;
        pair2++;
    }
    if (i > x) {
        x--;
        pair2++;
        pair3++;
    }
    if (i % 2 ==1) {
        x *= 2;
        pair3++;
    }
    assert pair1 < 2;
    assert pair2 < 2;
    assert pair3 < 2;
    return x;
}
```

Figure 24: Applying the naïve encoding to `example2`.

Otherwise, the pair truly are exclusive.

Transforming the Source Code To modify the source code and insert the necessary statements, we developed a program using Spoon [9], which is an AST analysis and modification library for Java.

First, it is necessary to identify series of `if` statements that could potentially be corrected to `else if` statements. Once identified, we then insert the necessary initialization, assignment, and assertion statements into the AST. These first two steps an be accomplished with a modification of the algorithm from Figure 19. The modified algorithm is found in Figure 28. Finally, the new AST is emitted as a Java source file and run against JDart.

```java
void example2Transformed2(int i) {
    int x = 0;
    boolean branch1 = false;
    boolean branch2 = false;
    boolean branch3 = false;
    boolean branch4 = false;
    if (i % 2 == 0) {
        x++;
        branch1 = true;
    }
    if (i < x) {
        x = 2;
        branch2 = true;
    }
    if (i > x) {
        x--;
        branch3 = true;
    }
    if (i % 2 ==1) {
        x *= 2;
        branch4 = true;
    }
    assert branch1 ^ branch2;
    assert branch2 ^ branch3;
    assert branch3 ^ branch4;
    return x;
}
```

Figure 25: Simplified encoding using Boolean variables, rather than counters.

Limitations The dynamic analyzer wasn't fully realized for several reasons related to JDart. These issues could potentially be solved with correct configuration, or even using another concolic execution tool for Java.

The first issue is related to JDart output. If JDart reports that all execution paths are OK – meaning the presence of an `if` statement that should be an `else if` – how do we determine where in the program it is? Unless the program only has a single pair of `if` statements, which is unlikely, it is not possible to know which pair of `if` statements should be made exclusive. It may be possible to configure JDart to give more detailed output, but it's not immediately clear how to do so.

The second issue is related to the program input. This approach was developed with an Algorithms & Data Structures course in mind, which means

```java
void example3() {
    if (condition1) {
        foo();
    }
    if (condition2) {
        bar();
    }
}
void example3Transformed1() {
    boolean branch1 = false;
    boolean branch2 = false;
    if (condition1) {
        foo();
        branch1 = true;
    }
    if (condition2) {
        bar();
        branch2 = true;
    }
    assert !(branch1 && branch2);
}
```

Figure 26: The first transformation under the final encoding scheme.

```java
void example3Transformed2() {
    boolean branch1 = false;
    boolean branch2 = false;
    if (condition1) {
        foo();
        branch1 = true;
    }
    if (condition2) {
        bar();
        branch2 = true;
    }
    assert !branch1 && !branch2;
}
```

Figure 27: The second transformation under the final encoding scheme.

that student assignments typically require collections of data to build into data structures, or to run algorithms on. JDart is best suited for programs with primitive data type input, rather than collections of data. It is not clear how to run JDart on a program that is expecting an array or list of data. A branch of JDart called JDoop was briefly discussed as a potential solution, but it was

```
For each code block c:
    IfStatements = new list
    For each statement s in c:
        If s is not an if statement:
            Cache IfStatements
            IfStatements = new list
        Else if s terminates with a branching control flow statement:
            IfStatements.Add(s)
            Cache IfStatements
            IfStatements = new list
        Else if s has a child else statement:
            IfStatements.Add(s)
            Cache IfStatements
            IfStatements = new list
            e = deepest else if child of s
            If e exists:
                IfStatements.Add(e)
        Else:
            IfStatements.Add(s)
    Cache IfStatements

For each cached list l:
    For i=0 to n=l.length:
        s = l[i]
        Insert ``boolean branch_i = false'' before l[0]
        Insert ``branch_i = true'' in s.Body
    For i=1 to l.length:
        s = l[i]
        Insert ``assert branch_i-1 ^ branch_i'' after l[n]
```

Figure 28: Algorithm to modify the source code by inserting new variables and `assert` statements.

not investigated.[5] It may still be possible to use this approach in a more basic computer programming course, if the programming assignments typically require primitive data rather than collections of data. It also might be possible to address this issue by using a different concolic execution tool altogether.

3.1.3 The Static Approach

This approach employs an SMT solver to test conditions for exclusivity. An SMT solver can be used to decide the satisfiability of logical formulas. If the conditions for two `if` statements are found to be unsatisfiable, then we can conclude that the conditions must be exclusive since no assignment of variables results in both condition expressions being true. The SMT solver we use is Z3, and since the pattern detector is written in Java, we use the Z3 Java bindings to create the Z3 queries [4].

To test two `if` statements for exclusivity, we need to construct a Z3 query where we `assert` the conditions of each statement, and then check for satisfiability. For example, if the code we wish to test is as seen in Figure 29:

```
if (x == 0) {...}
if (x >= 0) {...}
```

Figure 29: A simple pair of `if` statements to test.

We'd need to construct the Z3 query seen in Figure 30. In this example, the

```
(declare-const x Int)
(assert (= x 0))
(assert (>= x 0))
(check-sat)
```

Figure 30: The Z3 query corresponding to the program in Figure 29.

output is `sat` since `x = 0` satisfies both conditions.

The translation from Java source to Z3 was accomplished using the visitor pattern. A visitor is a Java class that recursively traverses the Java AST and converts the kinds of expressions found to Z3 expressions. In developing the conversion process, we found there are 4 types of expressions that could be converted from Java to Z3. They are Boolean, integral (int, long, char, etc.), floating-point, and objects. Boolean expressions were naturally the easiest to convert to Z3. Integral expressions correspond to the Int type in Z3, and floating-point expressions correspond to the Real type. Operations over these expressions are simple to convert, however the various forms of numeric constants permitted by the Java grammar required thoughtful parsing. Object expressions were simply treated as Z3 Int expressions. While it might be possible to develop a Z3 sort for the objects encountered in a Java program, it seems sufficient to treat objects as numbers since objects in Java reduce to numeric addresses in memory. For example, testing object equality in Java is the same as testing the object's address for equality. This means that the condition expression in `if (obj1 == obj2)` can be translated to the Z3 query in Figure 31.

```
(declare-const obj1 Int)
(declare-const obj2 Int)
(assert (= obj1 obj2))
```

Figure 31: Objects in Java are treated as `Int` in Z3.

Java expressions that involve some indirection, such as array accesses, object member accesses, and method calls, are treated as variables in Z3 with the given type. For example, if we have an `int[] a`, the Java code `if (a[4] < foo.bar().baz())` corresponds to the Z3 query in Figure 32.

```
(declare-const |a[4]| Int)
(declare-const |foo.bar().baz()| Int)
(assert (< |a[4]| |foo.bar().baz()|))
```

Figure 32: In Z3 syntax the `|` characters denote symbol literals, so `|a[4]|` and `|foo.bar().baz()|` are the names of `Int` variables.

Note that it's safe to assume `a[4]` and `foo.bar().baz()` have the same type, since if they did not have the same type, Java's typechecker would fail during compilation. If the code doesn't compile then it has greater issues than whether or not it contains novice structure.

Limitations Due to limitations in the AST library for PMD, sometimes type information cannot be resolved on more complicated expressions. For example, in the expression `foo.bar().baz()`, PMD maintains the type of `foo` but not for `.bar()` or `.baz()`. This poses a challenge since `.baz()` defines the resulting type of the entire expression. In this case we treat the Z3 variable as an `Int`.

As mentioned above, this poses no problem if `.baz()` is an object. If it's a Boolean, then we lose some precision, and this could cause the detector to miss some exclusive statements.

For example, consider the Java code in Figure 33:

```
if (foo.bar() != foo.baz() && foo.baz() != foo.qux()) { ... }
if (foo.qux() != foo.baz()) { ... }
```

Figure 33: PMD maintains the type of `foo`, but not any of its members.

Since there's no type information here, the visitor will assume that the type each of these function calls is an `Int`. This means that Z3 will find that the corresponding Z3 query (Figure 34) is satisfiable. However, if these functions are actually Boolean, then it should not be satisfiable since there are not 3 distinct Boolean values.

```
(declare-const |foo.bar()| Int)
(declare-const |foo.baz()| Int)
(declare-const |foo.qux()| Int)
(assert (and (distinct |foo.bar()| |foo.baz()|) (distinct |foo.baz()| |foo.qux()|)))
(assert (distinct |foo.qux()| |foo.bar()|))
```

Figure 34: Unknown types are assumed to be `Int`.

There are a few workarounds for this issue. First, in some cases we may be able to infer the type of an expression. For example, expressions using arithmetic operations such as `+`, `-`, `*`, `/` can be assumed to be numeric types. Expressions using Boolean operations such as `&&` and `||` can be assumed to be Boolean expressions. Also, we can assume that expressions will pass typechecking, so knowing or inferring one part of an expression is enough to infer the entire expression. For example, in Figure 35 the expression `foo.bar()` may not have type information available, but since `b` is a local variable, we know its type is Boolean. Therefore, we can infer that the expression `foo.bar() != b` is Boolean as well.

This sort of type inference could be extended to apply to other expressions as well. In the example above, we can also infer that `foo.baz()` is also a Boolean. However this extension is not yet implemented. In fact, the detector in its current state would interpret the condition at line 2 as a Boolean expression, and the condition at line 3 as an integer expression.

```java
void example1 (boolean b) {
    if (foo.bar() != b) { ... }
    if (foo.bar() != foo.baz()) { ... }
}
```

Figure 35: In some cases we can apply limited type inference.

As a special case, we always treat the `.equals(Object)` method as a Boolean since it is defined as such for every object in Java.

Some uncommon Java expressions are currently not supported for conversion to Z3. For example, bitwise expressions are challenging to convert since they can be applied to any integral Java type, but in Z3 they can only be applied to BitVector types. However in the batch of 2420 submissions being studied, there were no instances of students of using bitwise expressions. The converter does currently support conversion for all kinds of expressions found in `if` statements in the 2420 batch.

Since the source code is only evaluated statically, this analysis assumes that any method calls do not produce side-effects that would change the result of a following if statement. This means it may produce some false positives, where the developer intended the `if` statements to not be exclusive. Consider the program in Figure 36.

```java
boolean b;

void example2 (int x) {
    if (b) { foo(); }
    if (!b) { ... }
}
```

Figure 36: Side-effects of `foo()` could modify the value of `b`.

If the variable `b` is not local to the `example2()` method, it's possible that the `foo()` method modifies the value of `b`. This means that the developer could have intentionally chosen an `if` statement instead of an `else if` statement, since they don't want to always skip over the second statement. If `foo()` is a locally defined method, it may be possible to test this with data flow analysis. However the possibility of using externally defined methods means that this approach would not always solve this issue. Notably, this sort of issue could be solved with dynamic analysis.

Since developer intent is a black box, at best the detector can point this out to the developer with messaging such as “These statements appear to be exclusive, consider using an else if statement or restructuring your code” versus stronger, more prescriptive statements that could be used by detectors of other patterns.

3.2 Discussion

Using techniques from software verification have been successful in expanding the kinds of patterns we can detect. Applying these techniques may even enable the detection of other novice patterns. In particular, the current detector for the ‘exclusive ifs’ pattern does not yet test if a final `if` or `else if` statement should be an `else` statement instead. Both the static and dynamic techniques seem applicable to this pattern. In addition, the pattern where students use extraneous cases with a general solution that already covers those cases [17] may also be detectable with dynamic analysis.

There remain some challenges with integrating these additional programs into a tool that students could use while they’re writing code. Ideally, such a tool would require minimal installation and maintenance, and would be seamlessly integrated with the user’s IDE. These challenges remain for future research.

4 Teaching Assistant Interviews

4.1 Methods

To investigate how Teaching Assistants interact with novice code structure, I interviewed four Teaching Assistants working for the School of Computing in the Fall Semester of 2020. I sent a recruitment email to a list of current teaching assistants, offering compensation for a one-hour interview about code structure. We had hoped to recruit 10 participants, but only 4 responded to the email. These students had a variety of backgrounds and experience. During each interview, they would be shown 6 samples of Java source code with various kinds of novice code structure. We would then discuss their opinions on each sample, what they liked or disliked, and what kinds of feedback they would give a student on each sample. The interviews lasted for roughly one hour, and the participants were compensated $15 for their time.

Five of the code samples came from the previously collected 2420 assignment submissions. They were identified using the pattern detector. The entire collection of assignments was analyzed by the pattern detector, which then produced a spreadsheet identifying each sample of code that was flagged for using novice structure. Some samples were selected for having multiple flags, and others were selected for having different kinds of novice structure from the other samples. Once selected, they were reviewed manually to ensure the overall selection would be interesting and diverse. Each sample was selected from a different assignment.

One of the samples was refactored to use expert structure, and both were modified to include subtle bugs. Both samples were used in the interviews. These were the only samples to include correctness errors. The other 4 were assumed to correctly execute the functionality they were meant to provide.
Finally, the samples were extracted from the original student submission file, and copied into a new file so the relevant code would fit on a single screen, to avoid needing to scroll during the interview. To avoid extraneous compiler errors and warnings, the minimal amount of additional code was added, largely off screen. The entire batch of 2420 assignments was de-identified prior to collecting samples for this interview. Extracting the samples into their own file was simply a means of making it easier to discuss during the interview.

The interviews were conducted remotely, using Zoom. First, I would ask them questions about their background as a student and as a teaching assistant. Among the 4 participants, there were 3 seniors, and one Master's student. Three participants were Computer Science majors, with one double majoring in Economics, and one Applied Mathematics major. The courses they've TA'd for include beginner courses such as COMP 1010 and CS 1410, all the way through senior-level courses such as CS 4150.

Next, I loaded the code samples using the Eclipse IDE, and shared my screen with the participants. This was meant to be similar to a remote office hours session, where the student would share their code from their IDE. I then asked them to review the sample and share their initial thoughts and opinions. If they made any comments about coding style or structure, I would ask them to elaborate. After they shared their initial thoughts, I would ask them to view the sample in the context of their role as a Teaching Assistant. What kinds of marks or feedback would they give if this were student code they saw during office hours, or during grading? Some of the samples had specific points of interest that I would discuss with them. These include comparisons between novice and expertly structured code, as well as debugging exercises. If these points weren't already brought up during our conversation, I would ask them about these points. Finally, I would ask them to share any additional thoughts they had before proceeding to the next sample.

4.2 Results

4.2.1 Participant Profiles

The participants recruited are all at least seniors, and have varying majors and teaching assistant experience. Each of the participants have been given pseudonyms:

Participant 1: Elspeth Elspeth is a senior who is double majoring in Computer Science and economics. They are considering pursuing a Master's degree. They have been a TA for 3 semesters, each time for COMP 1010 "Programming for All 1." This course is meant to be an introductory programming course for students not pursuing a degree in Computer Science, and is taught in Python.

Participant 2: Gideon Gideon is a senior who is majoring in Applied Mathematics. They have been TA'd for 3 semesters, twice for CS 1410, "Object Oriented Programming," and once for CS 4150, "Algorithms." "Object Oriented Programming" is an introductory course for those pursuing a major or minor degree in Computer Science. Some prior programming experience is expected, either through a prerequisite course, or a proficiency test. It is taught using Java, or C++. “Algorithms” is a senior-level course that expects proficiency with programming fundamentals. While not the primary focus, some versions of this course include a programming component that can be completed in a number of programming languages.

Participant 3: Liliana Liliana is a Master’s student in the Computer Science program. They have been a TA for 4 semesters. Once for COMP 1020, “Programming for All 1,” once for CS 3505 “Software Practice II,” and twice for CS 1410. “Programming for All 2” is follow-up course to COMP 1010. “Software Practice II” is a junior-level course that is meant to teach software engineering best practices. It’s taught in C++, and while it may be some student’s first experience with the language, teaching C++ is not the primary focus of the course.

Participant 4: Jace Jace is a senior majoring in Computer Science. They’ve TA’d twice for CS 2420 “Algorithms and Data Structures,” and once for CS 4150.

4.2.2 Code Sample 1

```java
10 public boolean checkout_(long isbn, String holder, int month, int day, int year) {
11     int i = 0;
12     LibraryBook book = null;
13     GregorianCalendar dueDate = new GregorianCalendar(year, month, day);
14     while (i < library.size()) {
15         if (library.get(i).getIsbn() == isbn) {
16           book = library.get(i);
17           break;
18         }
19         i++;
20     }
21
22     if (book != null) {
23         if (!book.checkedOut()) {
24             book.checkOut(holder, dueDate);
25             return true;
26         }
27     }
28   return false;
29 }
30
31 public boolean _checkout(long isbn, String holder, int month, int day, int year) {
32     LibraryBook book = null;
33     GregorianCalendar dueDate = new GregorianCalendar(year, month, day);
34     for (int i = 0; i < library.size(); i++) {
35         if (library.get(i).getIsbn() == isbn) {
36             book = library.get(i);
37             break;
38         }
39     }
40
41     return book != null && !book.checkedOut() && book.checkOut(holder, dueDate);
42 }
```

Figure 37: The code sample shown to the interviewees.

Overview The code sample seen in Figure 37 presents two versions of the same method to check out a book from a library system. The first method, `checkout_`, is the original student code verbatim, and uses novice code structures. The second method, `_checkout` is refactored to use more expert style. Line 14 uses a `while` loop that should be a `for` loop, and line 22 exhibits two kinds of novice style. First, it uses nested `if` statements that could be collapsed into a single `if` statement using an `&&` operator. Second, it returns literal Boolean values, rather than the result of an expression.

Participant 1 Data

1. Elspeth: I like the second one more, [the expert-styled implementation] just ‘cause it’s fewer lines of code, but the `if` statements from line 22 to 27 make it more understandable.

   Interviewer: You prefer this one, [the expert-styled implementation] ‘cause it’s fewer lines but you actually prefer, this condition [line 22] because it’s easier to read?

   Elspeth: Yeah

   Interviewer: So what would like an ideal version of this method look like? Like, to your preference.

   Elspeth: It would include the `for` statement from the bottom one, but then the conditional check from the top one.

   Interviewer: OK, why don’t you like the `while` statement?

   Elspeth: Because it should be a `for` loop ‘cause it’s incrementing a counter.

2. Interviewer: So let’s say I’m the student: “Why should I use this different structure? My code works just fine. Why should I redo it?”

   Elspeth: Just because, one: it’s fewer lines of code. It’s like 2 fewer lines ‘cause you won’t have to instantiate. You don’t have to increment on separate lines. I think it’s good practice to use a `for` loop when you should use a `while` loop in other cases.

   Interviewer: How come? What makes it a good practice?

   Elspeth: I once read this in *Cracking the Coding Interview*, that you should use `for` loops when you increment. And I guess I just took that as a standard.

3. Elspeth: I would say that line 41, was kind of hard to follow. If like, this was having an issue, it would be harder to figure out what line had the problem, or like what specifically had the problem. Because on line 41

there are three different things going on, so it could have been one of the checks that are having issues.

Interviewer: So you would want them to break out line 41 because it would be easier for debugging?

Elspeth: Yep.

Participant 1 Analysis In line 1, Elspeth expresses a mixed agreement with the research team about expert and novice style. While they agree that the loop at [line 14] should be be replaced with a `for` loop, they specifically state that the novice-styled `if` statement is easier to understand. In addition, in line 2 they also state the expert version would be more difficult to debug, if one of the three conditions was incorrect.

In line 2, Elspeth states they prefer the `for` loop since it's fewer lines of code, and that it's a good coding practice to use a `for` loop when incrementing a counter. Their source on why this is a good coding practice is an interview preparation book, not any instruction they received at the University. The reasoning also goes against conventional reasons of when to use a `for` loop. Websites such as Khan Academy [7] and StackOverflow [16] state that `for` loops should be used when the number of iterations is known at run time. It is unclear if this student did not receive this same instruction at the University, or if they did not retain it.

In line 3, we see Elspeth's reason for preferring the novice style is a practical concern. If three Boolean variables are on a single line, a debugger would step over the entire line at once, rather than showing the evaluation of each variable. This would make it more difficult to understand which condition is faulty and causing unexpected control flow. Debugging tools typically have the ability to evaluate expressions during execution, but this takes additional work. In contrast, when the `if` statements are nested like this, the value of each condition can be immediately understood during debugging.

Participant 2 Data

4. Gideon: So they both make sense. The top one [novice-style] might be easier to understand if you're a beginner, but the bottom one [expert-style] is slicker from a, a Boolean logic standpoint.

5. Gideon: I mean, in my opinion, I like the second one better because `for` loops are kind of built for this sort of incrementation idea, where you're just going to increment `i`, to get through the entire library. That's kind of what `for` loops are for, so it makes sense to use a `for` loop instead of a `while` loop there. Save some lines, save some hassle, etc, etc.

6. Gideon: And then, on line 41 ... Yeah I think this makes more sense too, because, it might be harder to understand up front, like there's more overhead to make sure you know what's going on, but, it's slicker, it saves a lot of code. And if you were to like comment it, you could just put one comment and then you're, you're done.

   Interviewer: What would you like to see from a comment?

   Um, I would probably just put a comment that says like- [to self] let me remind myself- just say like, "if the book is in the library and is not checked out, check it out." That just kind of guides the person reading it to know what they're looking for.

   But I mean, all in all I think either implementation [of the `return` condition] is fine. If you're going to do the top one, you should definitely like, comment it because it's a little confusing, but- I mean just a little lengthier, even if it's not that confusing.

Participant 2 Analysis Gideon expresses a preference for the expert implementation of the method, because they prefer the use of a `for` loop in this context over using a `while` loop. In line 5, they agree with Elspeth that `for` loops should be used along with incrementation, and that it's preferable to save a few lines of code.

In regards to the `return` condition, they state that either version is 'fine'; the expert version is more 'slick,' but the novice version may be easier to understand for beginners. So much so, that it would be best to include a comment describing the return statement (Figure 38):

```
41 // if the book is not in the library, and is not checked out, check it out
42 return currBook != null && !currBook.getInOrOut() && currBook.checkOut(holder, cal);
```

Figure 38: While Gideon doesn't necessarily prefer the single-line return statement, they'd prefer to add this comment if it were used.

It is interesting to note that their comment reads like a transliteration of the code syntax to English. This suggests that the meaning of this `return` statement is not immediately accessible to the participant, and it's preferable to have the translation available in a comment. While what constitutes a good code comment can be subjective, it's not clear that this comment adds any clarification beyond this transliteration.

Participant 3 Data

7. Liliana: I guess, I like the second [expert] way's implementation better, because I think it's more clean. One thing that I would give for the very first `checkOut()` method, is I don't favor `while` loops. I think they're a little bit more harder to understand, is what I would generally tell students because, um, I think it's a little bit harder to track values in that way, but it doesn't make it any less correct. I think the biggest difference that makes the second method preferable to me, is the final `if` checks, where you return true or where you return false. Obviously to new coders, or like, students that are new to CS, the second [expert] approach wouldn't be intuitive, so I would understand why a student would do the first [novice] method's way, but in either case I think the second method [expert] is more readable and easily to be understood.

8. Interviewer: So in the class that you're currently TA'ing for, what is expected of you in terms of giving feedback to the student about these sorts of things?

   Liliana: So Professor Pseudonym is the professor that I'm currently TA'ing for, and he's really good at making us TA's give in-depth feedback to students every time we grade it. So rather than taking 2 or 3 hours to grade assignments, it takes north of 4 hours to grade about 30 assignments, and it's because we make comments for each individual so that it's unique to everyone. And so for this we grade according to the coding style, and how they can fix their code to learn to do things a little bit more concisely, and make sure that it's still readable.

   Interviewer: When you say "fixed" do you mean fix correctness issues, or fixing style issues?

9. Liliana: Fixing style issues. If I were given, for example, the method on the top, I would have made a comment on that final `if` statement, saying

this can be done in a single-liner, and let's think through the logic of that. Because what we're returning is a true or false return statement, and `if` statements always return a true or false, so we can fall back on that functionality. So that's what the type of comment I would give to this kind of submission. Another thing is, personally I would comment on this `while` loop, because `for` loops are there for a reason, and we always- personally with my history of TA'ing, whenever students learn about `while` loops, they've also learned about `for` loops. Either close to each other, or at the same time. So they should be able to understand `for` loops initialize these counting, incremental variables in a single-liner, and I would assume that they should know that at this point. So having a `while` loop with a counter on the outside, not really the best way of representing this counter. Because `i` in this case is a counter, so I would just recommend using a `for` loop because `for` loops do that in a single line and so it makes it more readable.

Participant 3 Analysis In line 7 Liliana states that they dislike the use of the `while` loop in this context. The reasons they give are that it's harder to understand, and that `for` loops initialize and increment their counter variable all in a single-line. Again, the reasons for this preference don't really relate to common knowledge about `for` vs `while` loops. They even seem to suggest that using `for` loops is always preferable.

They also agree with the expert version of the return pattern. They are the only participant to express a strong preference for the expert version of this method, the other participants either prefer the novice version, or they are ambivalent about the two approaches. In addition, they recognize that a novice programmer might not think to use the expert version, or that they may not consider the expert version to be as intuitive.

In line 8 the participant explicitly states that giving detailed, individualized feedback about coding structure is very time consuming, even suggesting that it can take at least twice as long to do so. This further confirms what was suggested by other research; students may not be receiving instruction about good code structure because of the time commitment required to give personal feedback.

In line 9 Liliana gives their reasons for preferring the expert styles, and the kinds of feedback they would give to a student. They say that "Because what we're returning is a true or false return statement, and `if` statements always return a true or false, so we can fall back on that functionality." It's not clear what this means, and if a novice programmer is unlikely to think to use the expert pattern, it's not clear that this would be sufficient to prompt them to use the expert pattern.

Liliana also states that "because `for` loops do that in a single line and so it makes it more readble." Here they seem to be equating fewer lines of code with improved readability. While it's not clear if this is true in the general sense, it may be the case that novice code has a tendency to be too verbose rather than too terse. If so, this comment from the participant would make sense, given their background as a TA for primarily beginner-level courses. If it is indeed the case that novice code tends towards verbosity, this would be an important consideration for designing a tool to assist students with improving their code structure.

Participant 4 Data

10. Jace: OK, so first one, using a `while` loop, for some reason, and then doing `i++`. Yeah, that's weird, definitely. I have actually seen, other students do that where they do a `while` loop they do- Like I have seen a couple different ones where they do a `while` loop, and `i++` like something that literally could just be a `for` loop. (pause) Which I think is *weird*.

    Interviewer: So you said you have seen this before? Or you have not.

    Jace: Yeah, I've seen like probably two or three 2420 students that like whenever I come to help them they always use `while` loops. And they'll write like- they'll be given like pseudocode for an algorithm and then turn into `while` loops. And then I'm like "This works so much better as a `for` loop!" But I guess it's still like, you know, obviously still operational right? But, it just looks weird. When you have something you're iterating on, you know?

    Interviewer: And did you say in which course was that? Did you have the students-

    Jace: Uh 2420. (Int: 2420? OK) Yeah yeah. Yeah I haven't seen any of that in 4150. (Int: OK) but I don't look at too much code in 4150, so.

11. Jace: Yeah, I mean that last part seems reasonable enough I guess. Let's compare it to the next one so- (pause) OK, yeah, I like the- I really like

`break` statements, so this is- I, I like this a lot more and then- (pause) Hm. I don't know about the Boolean, though. The Boolean is a lot, so-

Interviewer: What do you mean by "the Boolean"?

Jace: The return statement. So like `book != null`, and it's not checked out, and also check it out. (Pause) I think it's a cool use of it, but- (pause) might be closer on the edge of it like, you definitely have to stop and look at it for second.

Interviewer: Do you have a preference for one method or the other?

Jace: Definitely the second one.

Interviewer: Even though you don't like the `return` statement as much?

Jace: Yeah, like, I think the `return` statement's like, it's on the cusp like I think it still is fine, but, I would probably prefer like an `if` statement there because you got like three things going on. But I much prefer the `for` loop with the `break`, rather than the `while` loop with the- the `while` loop looks strange to me.

Interviewer: So you said you prefer an `if` statement here? What would it look like? Something like the implementation here [in the novice version of the method]? Or would it look different?

12. Jace: I'd probably do both of those in the same `if` statement. And then, return- (pause) I guess `checkOut()`- `checkOut()` returns true only if it actually checked it out?

    Interviewer: Right

    Jace: So yeah, I would just say `if(book != null && !book.checkedOut()`, then I would return `checkOut()`. And then I'd just return false at the end.

13. Interviewer: What feedback would you give based on the expectations of your role as a TA in that course?

    Jace: I think if I saw multiple instances of a `while` loop being used where it didn't really need to be- Or I guess even just this one I'd mention like- "Hey I'm just wondering why you're using a `while` loop. It's just like, harder to understand what's going on, you know, rather, than like a `for` loop." 'Cause when see a `for` loop you can see it's iteration, and so it's strange to see a `while` loop in this scenario where like, you're just doing one variable iteration, right? So I think I would probably say something that even though- Like again, of course it's totally fine to keep it. I'd just be like "Hey, by the way, like a `for` loop would probably be better here."

```
if (book != null && !book.checkedOut()) {
    return book.checkOut(holder, dueDate);
}

return false;
```

Figure 39: Jace's preferred implementation.

Participant 4 Analysis In line 10, Jace agrees with the other participants in saying that they disagree with the use of a `while` loop in the novice example. In particular, they state that it looks 'weird.' Jace also comments that there were specific students in 2420 who would commonly use that pattern. They also note that they didn't notice this pattern as much in 4150. Either this is a result of students abandoning this pattern by the time they're seniors, or this is because Jace didn't view as much student code in 4150. This poses an interesting question about the persistence of the use of patterns by individuals. Is this a pattern that students are likely to learn on their own? Are some patterns more persistent than others? Are students consistent in their use of novice or expert structures?

In line 11, Jace states that the expert return pattern is 'cool,' but not immediately comprehensible. Their comments are similar to Elspeth's, saying that there's "three things going on." However, the strangeness of the `while` loop is so great, they still prefer the expert method. Jace's reaction to the expert pattern is also similar to Gideon's, which they called 'slick.' Both participants express a kind of admiration for single-line return statement, in spite of their suggested changes to make it easier to understand.

Jace suggests in line 12 to combine two of the conditions into an `if` statement, but returns the value of the `checkOut()` method, or literally `false` otherwise (Figure 39). This suggests that Jace may conceptually separate the `checkOut()` method since it's performing an action, whereas the conditions `book != null` and `book.checkedOut()` don't modify any program state. Since they specified “three things going on” it may also be the case that their preference is to break up multiple conjunctions in this manner.

4.3 Discussion

Given that the TA’s recruited for these interviews had a variety of experience and backgrounds, it’s no surprise to see their varied opinions with regards to code structure. There were as many opinions about the return structure as there were participants. Only one of the participants expressed a strong preference for the expert return pattern, one TA preferred the novice pattern, and the other two had varying degrees of ambivalence. Liliana indicated the professor she worked for placed educating expert structure as a priority, and it could be the case that the TAs will care as much about structure as they feel they’re required to. On the other hand, Elspeth preferred the novice style for purely practical concerns. Namely, that having the conjunction broken out into separate lines makes it easier to identify errors when using a visual debugging tool. This shows that at least for some patterns, teaching assistants may actually prefer novice style.

However there were some common threads throughout each of the interviews. All of the participants agreed that the `while` loop looked ‘weird’ or was otherwise undesirable. They also agreed about the reason for preferring a `for` loop: that for loops should be used for ‘iteration’ or ‘incrementation.’ However this is strange since `while` loops can also be used to iterate, and could be controlled by incrementing a variable. It’s also interesting that none of the participants gave examples of when to prefer a `while` loop over a `for` loop. This could be because the students lack experience with use cases where `while` loops are more appropriate, or they simply always prefer to use `for` loops. It may also be the case that the participants lack internalized reasons for their preferences. If this is the case, we may not be able to totally rely on TA experience when it comes to developing feedback for a pattern detector.

Another common feature about the interviews is the lack of discussion about student comprehension. For the most part, they made no comment about any implications the novice structure might have about the author’s understanding of the program. Liliana indicated she understood why a novice programmer would use the novice return pattern, but this was the only mention of identifying with student’s coding preferences. The `while` loop example was so alienating, most of the participants expressed confusion about why someone would write code like that. Jace noticed a few students from the 2420 course who would frequently use this pattern, but makes no comment about why they might do so. These students are regarded almost as a mystery. This suggests that there is some disconnect between novice programmers and TAs. Even though the TAs are students themselves, they may have advanced far enough that they're unable to place themselves at the level of understanding of a novice programmer.

5 Conclusions

Let us consider the research questions posed in section 2:

Are teaching assistants more likely to agree with experts or novices with regards to code structure?

It seems that at best, we might be able to assume the 'average' teaching assistant is an intermediate programmer. Teaching assistants are likely to have more experiences than the students they're assisting, so it follows that they're not quite novices. As demonstrated by how consistently the participants reacted to the novice `while` loop pattern, TAs are likely sufficiently advanced to agree with certain patterns.

On the other hand, given that the teaching assistants are still students, it follows that it would not be correct to assume they're expert programmers yet. This is shown by their varied reactions to the novice return pattern. In addition, their reasoning for preferring one pattern or the other was often vague and poorly articulated. This also suggests TAs haven't wholly mastered certain coding conventions.

This makes sense given the variety of backgrounds and experience that teaching assistants have. Instructors who wish to emphasis expert coding patterns would likely need to ensure their teaching staff are all on the same page. They shouldn't necessarily expect that their TAs will already agree with them or have the same expectations when it comes to coding structure. However, TAs are capable of identifying sophisticated coding patterns, even if they're hesitant to employ them, or unable to articulate why one pattern is preferable over another.

How do teaching assistants communicate with students about code structure?

It is likely that teaching assistants only give feedback about coding structure to the extent that they're required to. Grading assignments is already a tedious and time-consuming endeavor. Being asked to give personalized feedback on programs that might even be functionally correct would be even more time-consuming. Spending 5 additional minutes to give feedback on 30 assignments to grade would add 2 1/2 hours to the time spent on grading.

During office hours teaching assistants may be willing to give feedback on coding structure, but likely only when asked by the student. It's unclear how frequently this occurs. In beginner courses, when students are more concerned with getting their code to compile and run than with improving their code style, it may be a rare occurrence indeed.

This also poses questions about the scalability of TA-generated feedback about coding patterns. Even if all the TAs in every programming course were giving style feedback on every assignment, if their feedback is poorly articulated or difficult to understand, would the students really be able to apply it? Would this be an effective use of the limited time available to instructors and course staff?

Can we use software to identify the novice structures that we're interested in?

Yes. Some novice patterns can be detected with the AST analysis that common linting and static analysis tools offer. Expanding the kinds of analysis at our disposal, such as SMT analysis and concolic execution, we are able to develop tools to automatically identify even more kinds of novice patterns.

Can we use teaching assistant experience to inform the design of a pattern detector so it gives informative, actionable feedback?

This is unclear. If this is possible, we would need to identify teaching assistants who have shown a sophisticated understanding of coding patterns, or have demonstrated particular success in educating the students they work with. It seems unlikely that we could arbitrarily select a teaching assistant and rely on their experiences to create an effective tool.

However this is not to say that teaching assistants shouldn't be considered at all. Since not all TAs agree with the use of expert coding patterns, it will be necessary to design a pattern detection tool with this in mind. The feedback or instruction it gives will need to be accessible to the TAs and students alike, or at least to the TAs at minimum. It would not be useful to employ a pattern detector whose feedback is ignored or dismissed by teaching assistants, since this attitude may be reflected by the students as well.

References

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

[2] Braun, V., and Clarke, V. Thematic Analysis. In *The Handbook of Research Methods in Psychology*, H. Cooper, Ed. American Psychological Association, Washington, DC, 2012, ch. 4, pp. 57–71.

[3] Butler, Scott. personal communication.

[4] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, p. 337–340.

[5] Dimjašević, Marko and Rakamarić, Zvonomir. https://github.com/psycopaths/jdoop, 2018.

[6] Keuning, H., Heeren, B., and Jeuring, J. How Teachers Would Help Students to Improve Their Code. In *ITiCSE 2019* (2019), no. July, pp. 119–125.

[7] Khan Academy. When do i use a for loop and when do i use a while loop in the javascript challenges? https://support.khanacademy.org/hc/en-us/articles/203327020-When-do-I-use-a-for-loop-and-when-do-I-use-a-while-loop-in-the-JavaScript-challenges-, 2018. Accessed: 2021-05-04.

[8] Luckow, K., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., and Raman, V. Jdart: A dynamic symbolic analysis framework. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2016), M. Chechik and J.-F. Raskin, Eds., vol. 9636 of *Lecture Notes in Computer Science*, Springer, pp. 442–459.

[9] Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience 46* (2015), 1155–1179.

[10] Pihrt, Josef. Roslynator analyzers. https://github.com/JosefPihrt/Roslynator/blob/master/src/Analyzers/README.md, 2021. Accessed: 2021-05-04.

[11] PMD Source Code Analyzer Project. Java rules. https://pmd.github.io/pmd-6.30.0/pmd_rules_java.html, 2020. Accessed: 2021-05-04.

[12] PMD Source Code Analyzer Project. Pmd source code analyzer project. https://pmd.github.io/, 2021. Accessed: 2021-05-04.

[13] Ren, Y., Krishnamurthi, S., and Fisler, K. What help do students seek in Ta office hours? *ICER 2019 - Proceedings of the 2019 ACM Conference on International Computing Education Research* (2019), 41–49.

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

[15] SonarSource. Unique rules to find bugs, vulnerabilities, security hotspots, and code smells in your java code. https://rules.sonarsource.com/java. Accessed: 2021-05-04.

[16] user747858. when to use while loop rather than for loop. https://stackoverflow.com/questions/6710601/when-to-use-while-loop-rather-than-for-loop, 2011. Accessed: 2021-05-04.

[17] Wiese, E. S., Rafferty, A. N., and Fox, A. Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated. In *Proceedings of the 41st International Conference on Software Engineering* (2019), ACM, pp. 84–94.