Readable vs. Writable code: A Survey of Intermediate Students' Structure Choices

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

Anna N. Rafferty
Carleton College
Northfield, Minnesota, USA
arafferty@carleton.edu

Jordan Pyper
University of Utah
Salt Lake City, Utah, USA
jordan.pyper@utah.edu

ABSTRACT

Since intermediate CS students can use a variety of control structures, why do their choices often not match experts'? Students may not realize what choices expert prefer, find non-expert choices easier to read, or simply forget to write with expert structure. To disentangle these explanations, we surveyed 328 2nd and 3rd semester undergraduates, with tasks including writing short functions, selecting which structure was most readable or best styled, and comprehension questions. Questions focused on seven control structure topics that were important to instructors (e.g., factoring out repeated code between an `if`-block and its `else`). Students frequently wrote with non-expert structure, and, for five topics, at least 1/3 of students (48% - 71%) thought a non-expert structure was more readable than the expert one. However, students often made one choice when writing code, but preferred a different choice when reading it. Additionally, for more complex topics, students often failed to notice (or understand) differences in execution caused by changes in structure. Together, these results suggest that instruction and practice for choosing control structures should be context-specific, and that assessment focused only on code writing may miss underlying misunderstandings.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; **Student assessment**.

KEYWORDS

Control structures, Novice programmers, Discourse rules

ACM Reference Format:
Eliane Wiese, Anna N. Rafferty, and Jordan Pyper. 2022. Readable vs. Writable code: A Survey of Intermediate Students' Structure Choices. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499413

1 INTRODUCTION

Instructors want students to write code that is not just functional, but also uses code structures that promote readability (that is, follow *discourse rules* [18]). However, teaching students to generate code using an appropriate control structure for a task is extremely difficult and time consuming [e.g., 8, 25]. Research has found that 90% of introductory students used inappropriate structures for a programming task [10], and structure problems persist into students' 4th year of a BS degree [4]. Why do students' code structures choices differ from their instructors'? Students and instructors may disagree about what makes code readable. Students may find it easier to generate alternative (rather than appropriate) control structures [17], perhaps because the alternative structures mirror their solution strategy [20]. Students may also avoid certain structures because they don't understand them.

Investigating these possible explanations for students' use of alternative code structures is a necessary first step toward better instruction on structure. We examine seven common patterns with control structures, within individual methods (e.g., loops, `if`-statements). Students learn these structures in introductory courses, but often violate their discourse rules, even in their 2nd and 3rd semesters [21]. If discourse rule violations stem from conceptual misunderstandings, the curriculum should address this. At our participants' university, instruction on using these structures appropriately is in the 1st semester curriculum but not a standard part of the later courses. Instructors may give feedback during office hours, or mention an egregious problem with an assignment during a lecture. However, in general, our students who violate discourse rules after the 1st semester are left to diagnose and treat themselves.

We examine students' understanding and use of appropriate structures via three tasks: writing, judgments of code readability and normative style, and comprehension. These tasks provide a holistic picture of students' skills: writing short methods (designed to target specific structures) allows us to assess the prevalence of discourse rule violations outside of a specific homework context. Asking students to choose which version of a function is most readable (to them) and which exhibits the best style (according to experts) can show discrepancies between knowledge of discourse rules and agreement with them. Finally, comprehension questions about execution and behavior with different structures demonstrate if students understand and attend to the impact of structure choices.

Our results, from administering the survey to over 300 students, show that many students find alternative structures more readable. We replicate a finding that students can identify appropriate structures even when they find them less readable [21]. We go beyond prior work to show that the relation between students' judgments of readability and the code that they write varies across control structure topics: for three topics, more than 45% of students wrote code that was not consistent with the structure they chose as most readable. Further, for more complex topics, code that followed discourse rules was harder for students to understand, and in many cases, students were inattentive to how structure choices impacted execution. These results suggest that students' reasons for using alternative control structures differ across contexts, and for all but the simplest topics, it is likely that instruction and practice is needed beyond the 1st semester. This instruction should target code reading and understanding in addition to writing.

2 RELATED WORK

Writing code with good style and assessing whether existing code meets style guidelines are core learning outcomes in the ACM CS2013 curriculum [1]. Discourse rules are a subset of style more generally [16]. While discourse rules are a core part of coding expertise, experts often know them implicitly, making it difficult to teach others [18]. Still, even when guidelines are provided within a course, students frequently write code that violates them [e.g., 11, 17]. Discourse rule violations can impede students' work with their own code. For example, duplicate code makes programs harder for students to modify and understand, even in a block-based language [7]. Structure in small blocks of code has been called semantic style [4], highlighting that structure choices may be indicative of conceptual understanding and thus particularly important to examine within CS education. Students' coding choices have been examined both in homework submissions [4], and by surveying students [21]. However, prior survey-based work had one one writing task [21] or a very small sample size [22]. We build on this work by directly examining the relation between code structure and conceptual misunderstandings in a large sample of students.

There is increasing interest in how to support students in programming with good structure, especially since providing individual feedback is time-consuming for instructors and often infeasible in large courses. Automated supports for structure either flexibly give hints and feedback on a wide range of implementations, but only work for specific assignments with a large corpus of prior submissions [2] or target a fixed set of possible errors. WebTA uses static analyses to give feedback and hints [19]. The stylistic anti-patterns that it targets were based on instructors' experiences with students' style errors and can be augmented for particular style contexts. Other work has explored generalizing feedback from experts or instructors to a wider range of programs [6, 9]. PyTA targets both syntactic and stylistic errors [13]. It uses existing static analysis tools (pylint), but provides enhanced explanations of the errors to help students identify problems; using PyTA was associated with fewer repetitions of the same error and faster error fixes [13]. AutoStyle [2] generates automated style hints that help students, but many students still struggled to act on the hints [24]. UglyCode interactively visualizes style consequences [15]. Our investigation complements this work by suggesting areas beyond code writing where students may need support, and also by identifying gaps in students' understanding that should inform instructional design.

3 SURVEY TOPICS AND ITEMS

To assess students' understanding of control structures, we build on the Readability and Intelligibility of Code Examples (RICE) survey, which targets seven control structures (topics) [21]. For each

```
public static String word2(String word) {
  if (word.endsWith("?")) {
    word = word.substring(0, word.length() - 1);
  } if (word.endsWith("!")) {
    word = word.substring(0, word.length() - 1);
  }
  return word;
}
```

Listing 1: The conditions in this method appear exclusive, but an input that ends in "!?" would execute both branches. A comparison method, `word1`, used an `else if`.

topic, one structure is the most *appropriate*. *Alternative* structures reflect common novice implementations. Each topic involves structures that occur within single methods, are taught in beginning programming classes, and require only a few lines of code:

T1 Returning a Boolean expression (with operators) vs. literals: Returning the expression is appropriate (`return x > 7`). An `if` statement returning `True` or `False` is alternative.
T2 Returning a Boolean expression (with method call) vs. literals: like (1), with a method (`return s.equals("a")`).
T3 Unique vs. repeated code within `if` and `else`: When some behavior is shared across all cases, it could occur once, outside the `if-else` (appropriate) or in both the `if` and `else` blocks (alternative).
T4 Only necessary cases vs. extraneous cases: Appropriate code handles all inputs parsimoniously. Alternative structures include extraneous cases for particular inputs that are already handled well.
T5 Array iteration with `for` vs. `while` loop: Iterating over an entire array can be done with a `for` loop (appropriate) or a `while` loop (alternative).
T6 Exclusive cases with `if-else-if` vs. `if-if` statements: Code with multiple exclusive cases can be written using a combination of `if`, `else if`, and/or `else` statements (appropriate) or with a sequence of `if` statements (alternative).
T7 && vs. nested `if` statements: When at least two conditions must be true before executing other code, they can be conjoined in a single `if` statement with Boolean AND (appropriate) or separated in nested `if`-statements (alternative).

The original RICE survey included judgments of the readability and style of different structures, comprehension items, and a code-writing task for one topic. Our expanded version of the RICE survey, building on [22], has five writing tasks (covering six topics), and revised judgment and comprehension questions. It also includes code editing and revision, to be reported on in future work.

We revised the code blocks that students judged for readability and style for 4/7 topics, either to examine more complex content or because there was not unanimous expert agreement on which block was best styled [21]. We verified which block was best-styled on each new item, with unanimous agreement from three instructors. Yet, experts may disagree on which structure is best, and understanding the range of opinions experts is an important step for future work. Similarly, the control structures discussed here were chosen because of their importance to our partner instructors: other structures may be more important in other contexts.

To more thoroughly explore students' understanding of the impact of structure on execution, we revised the original RICE comprehension questions. These were all multiple-choice questions about a single code block of the form "What is the output for this input?". We dropped Boolean returns comprehension questions due to a ceiling effect in [21] and to constrain survey time. We then replaced some questions about single methods with questions comparing two or more methods, in three flavors: (1) for *Exclusive cases with* `if-else-if` *vs.* `ifs`, identify which method performs more comparisons for a given input (e.g., Listing 1), (2) identify if the methods always have the same output given the same input (e.g., Listing 2), and (3) a follow-up to (2) asking for an input that would produce different outputs across the methods, and what each output would be. These questions assess how well students differentiate the functionality of different structures, a which could aid in choosing appropriate structures in a variety of contexts. We retained the original input-output questions for the *Only necessary cases* topic, and added a question about alternative structure code for *&& vs. Nested* `ifs` that asked (1) did the code accomplished task (in general) and (2) the output for a given input.

4 SURVEY METHODS AND HYPOTHESES

We tested seven hypotheses using the revised survey:

H1 For four topics (both Boolean returns, only necessary cases, and unique code in `if-else` blocks), at least 20% of students would indicate an *alternative* structure was more readable than the appropriate structure.

H2 For the remaining three topics, fewer than 20% would indicate that an alternative structure was more readable than the appropriate structure

H3 Students would recognize the appropriate structure as best styled more often than they would find it most readable.

H4 For T1 and T4 (returning a Boolean expression with operators, only necessary cases), at least 20% of students would mis-identify an *alternative* structure as appropriate.

H5 For the remaining five topics, less than 20% would mis-identify an *alternative* structure as appropriate.

H6 Consistent with past work [21], students' accuracy on code comprehension with different structures would not be predicted by which structure they said was most readable.

H7 Students' readability preferences would predict their code writing structure (consistent with [21] for Boolean returns and extending to five additional topics).

These hypotheses, and our 20% as a cutoff, are consistent with [21]. We use 20% anticipating that an instructor would want to address issues that affect at least 1 out of 5 students. These hypotheses were pre-registered (https://osf.io/32wuv). Beyond testing these hypotheses, our survey provides data that more fully describes the patterns in students' understanding, preferences, and use of control structures.

4.1 Methods

Participants were recruited from two intermediate courses in the CS major: a second semester course in data structures and algorithms, and a third semester introduction to software engineering (taught by the same instructor). The instructor publicized the survey to all

```java
public static String combo1(int[] nums1, int[] nums2) {
  String combinations = "";
  for (int i = 0; i < nums1.length; i++) {
    for (int j = 0; j < nums2.length; j++) {
      combinations += " (" + nums1[i] + ", " + nums2[j] + ")";
    }
    combinations += "\n";
  }
  return combinations;
}

public static String combo2(int[] nums1, int[] nums2) {
  String combinations = "";
  int i = 0;
  int j = 0;
  while (i < nums1.length) {
    while (j < nums2.length) {
      combinations += " (" + nums1[i] + ", " + nums2[j] + ")";
      j++;
    }
    combinations += "\n";
    i++;
  }
  return combinations;
}
```

Listing 2: The nested loops appear similar, but the `while` loop does not reset the inner counter. A comprehension item asked: given the same input, do both functions always give the same output?

students, and offered extra credit for completion. Students could access the survey and receive extra credit without participating in the research. The study was approved by our IRB (# 00114708). 328 participants consented to the research and skipped no more than one question per section.

Students had one week to complete the online survey, which had five sections: (1) code writing, (2) style and readability preferences, (3) comprehension, (4) code editing, and (5) code revising. For style and readability, students were shown 3-4 code blocks and first asked to select which was most readable and then which was best styled. To control for ordering effects, students were randomly assigned to forward or reversed question order within each question block except editing and revising; later analyses collapse across orderings.

4.2 Data coding for writing responses

First, functionality was evaluated via automated tests. For code that did not compile, some manual edits were permitted to attempt to fix errors while preserving intent (e.g., `array.length()` could be modified to `array.length`); allowed changes were pre-registered and refined as we found new errors (https://osf.io/gzxmf/).

Regardless of compilation, structure was also evaluated, with some minimum functionality required (e.g., for `&&` *vs. Nested* `ifs`, code must evaluate at least two conditions). Responses meeting these requirements were coded as following or violating discourse rules for the targeted topic. Categorization guidelines were based on the appropriate structure for each task, refined based on student responses, and verified by the instructor (https://osf.io/3r2ax/).

Array iteration with `for` *vs.* `while` *loop* (T5) was evaluated via regular expressions. Boolean returns (T1, T2) were evaluated with regular expressions and hand-checked for ternary operators or insufficient functionality (e.g., always returning `false`). For code that compiled, we identified `&&` *vs. Nested* `ifs` (T7) with static analysis

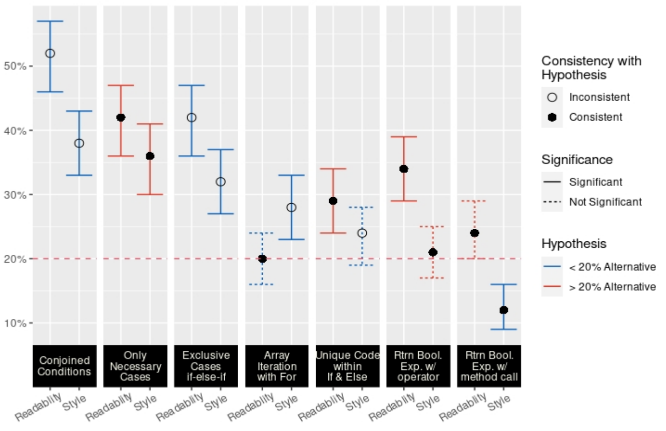

Figure 1: Percentage of students who chose *alternative* structures as most readable and best styled. Point represents estimate and error bars are the confidence intervals.

and hand-checked. For all other topics (and && *vs. Nested* `if`s for non-compiling code), two coders scored 29-30 non-blank responses, with Cohen's κ of .78 for *Only necessary cases* (T4), and > .9 for && *vs. Nested* `if`s (T7), *Unique vs. repeated code within* `if-else` (T3), and *Exclusive cases with* `if-else-if`*s vs.* `if-if` (T6) (κ =.60-.79 indicates moderate agreement; .80-.89, strong; and $\geq$ .90, almost perfect [14]). For T3, a third coder achieved pair-wise κ > .8 with the two original coders. After resolving disagreements, one coder scored remaining responses.

5 RESULTS

5.1 Style and readability choices

Students' judgments of what structures were most readable differed markedly from those chosen by instructors as most appropriate: for an average of 2.4/7 topics, students chose an alternative structure as most readable. z-tests supported Hypothesis H1 for all four topics where we predicted that $\geq$ 20% of the population of 2nd and 3rd semester CS students would select an alternative code block as most readable, replicating prior results [21] (Figure 1). However, for the other three topics, H2 (< 20% of students would select an alternative code structure as most readable) was not supported. Rather, z-tests for *&& vs. Nested* `if`*s* and *Exclusive cases with* `if-else-if` *vs.* `if`*s* topics were significant for $\geq$ 20% of the population choosing an alternative code block as most readable (p < .0001 for both). This indicates that a large minority of students find violations of discourse rules to be more readable than adherence to them.

For recognizing experts' preferred style, z-tests supported H4 for one of the two topics where we predicted $\geq$ 20% of students would say an alternative code structure had the best style (p < .0001 for *Only necessary cases*, not significant for *Returning a Boolean expression with operators*). For the five topics in H5 where we predicted that < 20% of students would choose alternatively structured code blocks as best styled, again only one was supported (Figure 1).

H3 was supported: students were significantly more likely to choose appropriately-structured code when asked what an expert would prefer prefer (vs. when asked which was the most readable). We tested H3 with logistic regression. The dependent variable was chosen structure (binary: appropriate or alternative), with question topic (T1-T7) and type (style or readability) as predictors, and a random predictor for student; β = 0.2186, $t(4584)$ = 6.33, p < .0001. Exploring the effect of course level, we found only one significant difference: students in software engineering (3rd semester) were more likely to prefer reading expert styled code than those in data structures and algorithms (2nd semester) (β = −0.15652, $t(2294)$ = 2.3514, p = 0.019).

Results for T7 are strikingly different from the original RICE survey [21], where about 90% of students thought conjoining conditions with && was both more readable and better styled than nested `if`s. Less than 65% of our sample thought so, likely because our revised survey use code blocks relying on short-circuit evaluation. See [23] for separate discussion of this result.

5.2 Comprehension of code

Comprehension questions revealed some misunderstandings for particular structures, but in general, these difficulties were not related to which code structure a student said was most readable (consistent with H6). For all topics, logistic regressions showed that students were not more accurate for the structure they said was more readable (dependent variable: correctness; predictors: code structure of item, structure selected as most readable, the interaction between these two, and a random effect for student).

For T4, *Only necessary cases*, students were less accurate at identifying the output for code with only necessary cases compared to code which included extraneous cases (77% vs. 86% correct; coefficient for alternative structure = .38, $t(2298)$ = 5.5, p < .0001). This was due to greater accuracy on inputs handled by the extra cases (typically, these inputs were length 0 or 1 arrays, with the general solution involving a loop): students answered 91% of these accurately when shown the alternative structure code, compared to 76% when shown the code with one general solution. Code structure made little difference for other inputs (79% correct for both structures). Accuracy on these "edge case" inputs for appropriately structured code suggests potential challenges with understanding loop entry and exit conditions: although these edge cases required fewer tracing steps than "normal" input, tracing was harder.

For T7, && *vs. Nested* `if`*s*, students were more accurate with nested `if`s (logistic regression with structure and condition order as predictors, with a random student effect; coefficient for nested `if`s = 0.61, $t(1308)$ = 9.4, p < .0001). One item presented three similar methods that checked the same conditions in two different orders (two with nested `if`s, one with &&). Only 52% of students identified how condition order affected short-circuit evaluation. Most students stated all three blocks were equivalent, suggesting that many students are inattentive to how small structural changes impact functionality; for more detail on conjoined conditions see [23].

For T6, *Exclusive cases with the* `if-else-if` *vs.* `if-if`, 63% of students incorrectly stated the two code blocks described in Listing 1 had the same functionality (z-test for $\geq$20% of students: p < .0001). A side effect within the first if-statement meant that some inputs for the `word2` method could cause both statements to evaluate to `true`; this would not happen in `word1` because of the `else-if`. When given an input that had different outputs for each method, and

Table 1: Responses for which method from Listing 1 executes more comparisons for some input, grouped by responses for functionality. Most students incorrectly stated they had the same functionality; most also realized the if-if code would execute more comparisons.

	Choice for more comparisons		
Do code blocks differ in functionality?	word1 [if-if]	word2 [if-else-if]	Same number
Yes: Correct (34% overall)	89% (100)	4.5% (5)	6.3% (7)
No: Incorrect (66% overall)	68% (147)	11% (23)	21% (46)

asked which code performed more comparisons, students were more accurate, but 25% still answered incorrectly (z-test for $\geq$20% of students: $p = .008$). Most incorrect answers stated the two methods performed the same number of comparisons, especially among students who answered the functionality question incorrectly (Table 1). For a pair of code blocks with truly exclusive cases, 98% of students correctly indicated that input/output functionality was identical (z-test for $>$ 80%: $p < .0001$), but 37% incorrectly identified which block performed more comparisons for given inputs (z-test for $>$ 20%: $p < .0001$). We thus see that a large minority of 2nd and 3rd semester students don't understand how a series of `if` statements differs from an `if-else-if` sequence, and their inattention persists even when pointed to specifics input. This gap may affect how they read and write code.

For T5, *Array iteration with* `for` *vs.* `while`, one item presented equivalent single loops, and 96% of students correctly identified that they had the same functionality. Another item presented the code in Listing 2, with doubly-nested loops. Although the loops appear similar, the `while` code does not reset the internal counter to 0; 83% *incorrectly* said the methods were equivalent (z-test for $\geq$ 20%: $p <$ .0001). Consistent with T6 and T7, these results suggest students are not particularly sensitive to the details of control structures and the impact of structure choices on likely bugs.

5.3 Code writing

For all but two topics, z-tests indicate that at least 20% of 2nd and 3rd semester students violate discourse rules (see Table 2). Except for T7 (&& *vs. Nested* `if`*s*), the proportion of students who wrote with appropriate structures was similar for code that passed all tests and for all code with sufficient functionality to evaluate structure. For T7, students were *much less likely* to use && if their code passed all tests (33% of fully functional responses used && vs. 62% of all responses with $\geq$ 2 conditions).

Students' selection of which code block was most readable predicted their writing style for 3/6 topics (H7 partially supported). As in past work [22], readability preferences predicted writing style for T1 and T2, *Returning Boolean expressions* (separate logistic regressions predicting writing structure, with readability preference as a predictor: Boolean expression with operators coefficient = 1.34, $t(313) = 5.4$, $p < .0001$; Boolean expression with method call coefficient = 0.77, $t(326) = 3.2$, $p = .002$). When students' writing and readability styles did not match for T1 and T2, it was more common for students to write with an alternative structure while still choosing the appropriate structure as most readable (Table 3). Students may thus like the appropriate structure but not always produce it. Following the lower levels of the SOLO taxonomy, these students may be directly translating English instructions into code [12].

T3, *Unique vs. repeated code within* `if` *and* `else`, showed mixed evidence for how readability preferences predicted writing structure: readability was predictive of writing style when all submissions were included (coefficient for readability preference = 0.93, $t(211) = 2.5$ $p = .01$), but not when including only submissions that passed all tests (coefficient = .13, $t(119) = .26$, $p = .79$). Only 37% of submissions passed all tests, and those submissions tended to be from students who selected the appropriate choice for readability (83%, versus 69% of all students whose code could be evaluated for structure). Yet, for both sets, the plurality of students chose the appropriate structure for readability and wrote with an alternative structure (58% for all, and 45% for code passing all tests; Table 2), and only a few students chose the alternative structure for readability but wrote with an appropriate structure (7.8% for all, 7.5% for code passing all tests). That is, when students' reading preferences differed from their implementation choices, they were more likely to prefer an expert choice (but not implement it) than the reverse, suggesting that preferring the expert choice is a precursor to using it. This aligns with prior findings that writing code tends to follow other programming skills [e.g., 12], although that work has not examined students' readability preferences.

For the remaining topics with writing tasks, readability preferences did not predict writing. For && *vs. Nested* `if`*s*, across submissions, 26% selected an alternative structure for readability and wrote with an appropriate structure, and a similar number of students showed the opposite pattern (Table 3). When including only fully functional code, more students said the appropriately-structured code was most readable than actually wrote that way. However, appropriately structured code was less likely to pass all tests.

For T4 (*Only necessary cases*) and T5 (*Array iteration with* `for`), students were more likely to write with appropriate structure than to identify appropriately structured code as most readable (Table 3). However, results from the comprehension questions for T4 (where students understood *alternative* code more easily for edge cases), suggest that some students who write appropriate code may not fully understand why the code works.

6 DISCUSSION

Overall, our results show that across topics, students are more likely to identify which structure follows discourse rules than they are to agree that this structure is more readable. Beyond this, students' understanding of control structures varies across topics, suggesting that a unified instructional strategy may not be feasible. Several themes in how code writing, comprehension (reading), and judgments of style and readability are related provide further insight into students' skills. For the simplest topics (T1 and T2, returning Boolean expressions), most students recognized appropriate structure, frequently found it most readable, and tended to write code that aligned with the structure they found most readable. When students' writing structure and readability choices disagreed, most students found the appropriate structure more readable while writing with an alternative structure, suggesting that instruction

Table 2: Proportion of responses with appropriate structure, by topic. *Passed all tests* has only functionally correct responses, while *Evaluated* includes all code whose structure could be categorized. p-values are for z-tests of ≥20% alternative structure.

	Writing: Passed all tests			Writing: All Evaluated for Structure		
Topic	n	% appropriate	p	n	% appropriate	p
T1 Return Boolean expression w/ Operator	312	57%	$p < .0001$	328	54%	$p < .0001$
T2 Return Boolean expression w/ Method Call	294	62%	$p < .0001$	328	59%	$p < .0001$
T3 Unique Code within `if` and `else`	121	46%	$p < .0001$	213	42%	$p < .0001$
T4 Only Necessary Cases	97	79%	$p = .44$	282	74%	$p = .008$
T5 Array Iteration with `for`	77	90%	$p = .983$	240	93%	$p = .999$
T7 Conjoined Conditions with &&	109	33%	$p < .0001$	318	62%	$p < .0001$

Table 3: Writing structure and readability preferences (for expert vs. alternative structure). A disagreement between how students write and what they prefer to read is common, varying from 29% to 66% of students, depending on the topic.

	Disagreement			Agreement		
	Write Expert	Read Expert	Total	Both Expert	Both Alternative	Total
Unique Code within `if` & `else` (T3)	8%	58%	66%	11%	23%	34%
Conjoined Conditions with && (T7)	26%	26%	52%	33%	14%	47%
Only Necessary Cases (T4)	33%	14%	47%	42%	11%	53%
Return Boolean Expression w/ Method Call (T2)	14%	25%	39%	45%	16%	61%
Return Boolean Expression w/ Operator (T1)	13%	21%	34%	42%	24%	66%
Array iteration with `for` (T5)	24%	5%	29%	69%	2%	71%

helping students to remember to write in an appropriate structure might be sufficient.

In contrast, students demonstrated less mastery of the other topics: for most of these topics, fewer students identified the appropriate structure and they were less likely to choose it as most readable. Students often did not notice how structure affected functionality (T6, `if-else-if` vs. `if-if`). For T5 (`for` vs. `while`), most students missed a bug common to the alternative structure (nested `while` loops). For both T7 && *vs. Nested* `if`*s* and T4 (*Only necessary cases*), students were generally more accurate on code-tracing questions with alternative rather than appropriate structures. For T3-T7, readability choices and writing structure were less aligned than for T1 and T2, but not in a consistent way: e.g., more students chose the alternative structure as most readable for T4 than wrote with it, suggesting that appreciating the readability of the appropriate structure is harder than creating it, but the opposite pattern held for T3 (*Unique vs. repeated code within* `if` *and* `else`). This interplay between identifying, using, and understanding appropriate structure means that, for all but the simplest structures, students will likely need carefully designed instruction and practice to improve; simply flagging discourse rule violations is unlikely to be sufficient and may fail to address misunderstandings about how structure impacts execution.

Threats to Validity. Since extra credit was given for survey *completion*, not correctness, students may not have tried their best, causing us to underestimate their abilities. Students' opinions on the style and readability of short code blocks may not reflect their judgments in more complex contexts, although given that some revised survey questions increased complexity (e.g., adding reliance on short-circuit evaluation) and preference for appropriate structures decreased, we believe more complexity may lead to increased affinity for alternative structures. We did not probe students' views of what "readable" means, asking only that they identify "Which one is most readable to you: which one makes it easiest for YOU to figure out what the code does?"; students may differ in their interpretation of readability in this context. Our students attend a college of engineering at an R1 university, so results may not generalize to other contexts such as liberal arts or community colleges.

6.1 Takeaways for instructors

A surprisingly high proportion of students had comprehension errors with loops, short-circuit evaluation, and `if-else` sequences. In courses where the survey was conducted, most reading/editing exercises were only on exams; the bulk of students' practice was weekly code-writing assignments. Since professional software development often involves reading existing code [3, 5], these results argue for integrating more code-reading practice, even at the expense of writing.

Especially by their 3rd semester, students are expected to follow the discourse rules noted here. Yet, at our participants' university (and likely at others), formal instruction or consistent feedback on structure is not given after the 1st semester. This study shows that students struggle beyond CS1, which may create matching struggles for the course staff. Poorly-structured code creates additional burdens for these staff, as it is more time-consuming for instructors and TAs to help debug. It also raises pedagogical dilemmas: should TAs help students find the bug, ignoring structure, or try to show students how good structure could have avoided the bug (which students may not be ready or willing to engage)? When course staff do provide in-depth feedback on structure, they can't tell if it sticks. These results strongly argue for formalizing repeated instruction and practice related to control structures in courses beyond CS1, and for exploring effective ways to teach structure.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) award SHF 1948519.

REFERENCES

[1] Association for Computing Machinery (ACM) and IEEE Computer Society. 2013. Curriculum Guidelines for Undergraduate Degree Programs in Computer Science by the Joint Task Force on Computing Curricula. https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

[2] Rohan Roy Choudhury, HeZheng Yin, Joseph Moghadam, and Armando Fox. 2016. Autostyle: Toward coding style feedback at scale. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*. ACM, 21–24.

[3] Cecil Eng Huang Chua, Sandeep Purao, and Veda C Storey. 2006. Developing maintainable software: The Readable approach. *Decision Support Systems* 42, 1 (2006), 469–491.

[4] 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*. ACM, 73–82.

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

[6] Elena L Glassman, Lyla Fischer, Jeremy Scott, and Robert C Miller. 2015. Foobaz: Variable Name Feedback for Student Code at Scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15*. 609–617. https://doi.org/10.1145/2807442.2807495 arXiv:13/02 [978-1-4503-1332-2]

[7] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[8] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating pedagogical code reviews into a CS 1 course: an empirical study. *ACM SIGCSE Bulletin* 41, 1 (2009), 291–295.

[9] Michelle Ichinco, Aaron Zemach, and Caitlin Kelleher. 2013. Towards generalizing expert programmers' suggestions for novice programmers. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 143–150.

[10] 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.

[11] Xiaosong Li and Christine Prasad. 2005. Effectively teaching coding standards in programming. In *Proceedings of the 6th Conference on Information Technology Education*. ACM, 239–244.

[12] Raymond Lister, Tony Clear, Dennis J Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, et al. 2010. Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin* 41, 4 (2010), 156–173.

[13] David Liu and Andrew Petersen. 2019. Static Analyses in Python Programming Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 666–671.

[14] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemica Medica* 22, 3 (2012), 276–282.

[15] K McMaster, S Sambasivam, and Stuart Wolthuis. 2013. Teaching Programming Style with Ugly Code. In *Information Systems Educators Conference*, Vol. 30. San Antonio, TX. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.400.9411&rep=rep1&type=pdf

[16] Paul W Oman and Curtis R Cook. 1990. A Taxonomy for Programming Style. In *Proceedings of the 1990 ACM Annual Conference on Cooperation CSC '90*. Washington, DC, 244–250. https://doi.org/10.1145/100348.100385

[17] Stewart D Smith, Nicholas Zemljic, and Andrew Petersen. 2015. Modern goto: novice programmer usage of non-standard control flow. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM, 171–172.

[18] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283

[19] 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*. ACM, 738–744.

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

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

[22] Eliane S Wiese, Anna N Rafferty, Daniel M Kopta, and Jacqulyn M Anderson. 2019. Replicating novices' struggles with coding style. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 13–18.

[23] Eliane S Wiese, Anna N Rafferty, and Garrett Moseke. 2021. Students' misunderstanding of the order of evaluation in conjoined conditions. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE.

[24] 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 4th ACM conference on Learning at Scale*. ACM, Cambridge, MA, 41–50.

[25] Michael Woodley and Samuel N. Kamin. 2007. Programming Studio: A Course for Improving Programming Skills in Undergraduates. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, USA) *(SIGCSE '07)*. ACM, New York, NY, USA, 531–535. https://doi.org/10.1145/1227310.1227490