# Understanding Similar Code through Comparative Comprehension

Justin Middleton, Kathryn T. Stolee
Department of Computer Science, North Carolina State University
{jamiddl2, ktstolee}@ncsu.edu

2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) | 978-1-6654-4214-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/VL/HCC53370.2022.9833117

***Abstract*—Any problem in code may have multiple solutions that differ in details large and small. Because modern software development is characterized by an abundance of searchable and reusable code, effective developers must be able to judge not only the meaning of new algorithms but also the differences between alternatives. Therefore, we use a multi-method study to explore how developers perform *comparative comprehension*—the cognitive activity of understanding how algorithms behave relative to each other.**

**To explore how developers compare code, we performed a controlled experiment with 16 developers in a mixed think-aloud and interview format and another 95 developers in a survey format. In this experiment, participants investigated whether a pair of code snippets would demonstrate equivalent behavior when run, controlling for differences in behavior, programming languages, algorithmic structures, and meaningful names. Overall, our results describe how comparison fits into learning, reviewing, and reusing code. Our task observations shed light on how developers move between code similarities at different levels—textual, structural, and schematic—when simultaneously inspecting multiple snippets. In our experiment, developers made more accurate conclusions about behavior given similar languages and structures, with names acting as additional evidence in interaction with other cues, but they also overestimated whether behavior is equivalent in many cases. From this, we identify challenges developers face in comprehending alternatives and we highlight opportunities to better support developers in comparison activities.**

***Index Terms*—comparative comprehension, program comprehension, code clones, human studies**

## I. Introduction

Being a software developer means managing options. If you are uncertain about how to write code for a problem, you can explore tutorials in a Google search, consult recommendations on Stack Overflow, or investigate code in context in codebases or open-source projects. These resources altogether contain a massive quantity of diverse applications, but despite this diversity, a significant amount of code contains similar features [1]. This repetition occurs for a variety of reasons: for example, similar sub-problems reoccur in different contexts [2], development communities share linguistic conventions [3], and developers directly reuse and modify each other's code [4]. This repetition from a diversity of sources means not only that a given algorithm may exist somewhere already and be waiting for discovery, but also that there may exist multiple similar but divergent alternatives that satisfy the problem. When there are multiple viable options, developers may have to move beyond *comprehending* individual programs into *comparing* alternatives for subtle differences.

Comparison is the cognitive activity by which people consider two or more things and identify features that they have in common and contrast those they do not. Any two things can be compared for more or less insight, but for software developers, code is the fundamental material of their work and the primary object of software comprehension research [5]. Given a task, different code designs have different implications—some may be faster, more readable, or more prone to error [6]—so the urgency of understanding code comparison is rooted in how it serves as fuel for decision making in design. Therefore, we ask the question: *how do developers comprehend the similarities and differences of code alternatives?*

To explore the process of comparison, which we also call *comparative comprehension*, we performed a multi-method study with a controlled experiment, think-aloud observations, and interviews to observe developer reasoning from multiple angles. In this paper, we focus on the case of code behavior, seeing it as one of the fundamental qualities which defines a program. Therefore, we observe how developers reason about code snippet pairs that differ in behavior as well as language, structure, and naming to explore how developers recognize the same principles at work in different designs. We also generalize how developers use comparative activities in order to perform many kinds of software development work.

Our data build a case for comparative comprehension in contexts like learning, reviewing, and beyond. Altogether, comparative comprehension builds on processes from singular comprehension, such as using experience and significant behavioral cues to reduce the work of bottom-up comprehension, but it introduces strategies for moving between alternatives like searching for and aligning structural analogues. However, these processes are vulnerable to error. Interview and survey participants achieve 73% and 65% correctness, respectively, in comparison tasks that involve identifying differences between alternatives; the think-aloud tasks reveal various types of mistakes developers make. Our survey results suggest that similar features help developers make more accurate conclusions about code behavior—similar algorithmic structures and shared languages have a positive impact on correctness, and meaningful names interact with other effects—but they also suggest that, in our case, developers overestimate the rate at

978-1-6654-4214-5/22/$31.00 ©2022 IEEE

which they think behavior is shared given two similar snippets. We discuss the implications these results have for theory and practice, such as in understanding the assumptions developers make in the contexts of multiple possible alternatives and how tools can support the discovery of significant differences.

Our research contributions include:

- A definition and qualitative description of comparative comprehension as a cognitive process, grounded in observations of task performance from 16 developers, and
- A survey and model from 95 developers, corroborating the relationship between syntactic similarities and the comprehension of semantic similarities in code snippets.

## II. Research Questions

To reiterate, given a landscape of software development with abundant yet repetitive code, the ability to make insightful distinctions between similar alternatives can make developers effective curators of code. We call this ability *comparative comprehension*, rather than just *comparison*, to emphasize it as an act of human cognition in contrast to comparative analyses performed by computational tools. Furthermore, rather than leave unexamined the notion that explicit comparison occurs because the conditions for comparison occur—more than one viable option exists for a tasks—we also want to explore if and in what contexts developers compare code in practice.

Measures of similarity are subjective and diverse, and it is difficult to say if any two algorithms are ever the "same" [7]. Nevertheless, few qualities are more fundamental to an algorithm than its *behavior*, or the values or states it produces. If the behavior does not reasonably address a specific task, then the developers are more likely to filter it out of focus rather than fully comprehend it [8]. Therefore, in this paper, we focus on behavior equivalence, or whether code will produce the same outputs for the same inputs, as our subject for comparative comprehension. In code similarity research, behaviorally similar programs are also called *type-4 clones* [9], and our use of "equivalence" is a more exact constraint on "similar." Although developers care about other comparable dimensions like complexity and readability, these concepts are more difficult to validate [10] compared to output behavior.

Lastly, whereas typical program comprehension studies often investigate questions of how code fits into a larger structure [11], our focus supposes similar program excerpts that are not designed to coexist in the same program at the same time. Instead, these snippets could ostensibly substitute for one another in the same structure, and they may imply design trade-offs which comparative comprehension can infer. This concept is similar to *variants* within the work of Srinivasa Ragavan and colleagues [12].

Therefore, we formalize these research questions:

**RQ1: In what contexts do developers perform comparative comprehension?** This research question grounds our definitions in real-world applications.

**RQ2: What strategies do developers use in comparative comprehension?** We look for the decisions and strategies that enable developers not only to make sense of each alternative but also to evaluate them as similar or different.

**RQ3: How do designs of code alternatives affect the accuracy of comparative comprehension?** Given that code design influences comprehension [13], we test whether dimensions of design similarity influence comparison too.

## III. Methodology

With a focus on equivalent behavior in code, we designed a controlled experiment and implemented two study designs to execute it: think-aloud tasks with reflective interviews, and a survey distribution. Think-aloud tasks allow for direct observation and personalized follow-up questions. Survey tasks, on the other hand, trade specificity of observations for broader distribution. Nevertheless, we implemented both of these designs with the same style of comparison tasks, which we describe in this section.

### A. Task Design

The core activity in this study's tasks is to **present developers with pairs of variably designed snippets and ask whether the snippets are behaviorally equivalent**. Given that there are two possible answers to this question (yes and no), our first independent variable (IV) is IV-1*: the ground truth of behavioral equivalence*. We call behaviorally equivalent pairs *clones* [9]; if not, then they are *non-clones*. The dependent variable is whether the developer accurately identifies the snippets as clones or non-clones.

To explore the impact of variable designs for RQ3, we identify three dimensions of design from literature that may influence comparative comprehension:

- IV-2*: Programming languages* {*same or different*}*:* Although each language has a unique syntax, they often share enough concepts to solve the same problems. Developers often work in multiple languages [14], but transferring knowledge between them is prone to error [15].
- IV-3*: Algorithmic structure* {*similar or dissimilar*}: The same tasks can be accomplished with different sequences of control and data features. For example, bubblesort and mergesort accomplish the same sorting task with different structures. Furthermore, different approaches in this way may represent different stages of learning and may be unequally vulnerable to error [6].
- IV-4*: Semantic content of names* {*meaningful or obfuscated*}: Function and variable names are a medium of communication and influence on comprehension [16], [17]. Rather than name similarity, however, we focus on whether names that suggest behavior improve comparative comprehension compared to obfuscated names.

As a baseline of similarity, we constrain our tasks to snippet pairs with identical type signatures and related semantics but not identical text. For related semantics, we group tasks into families for which all snippets can be accurately described by brief, natural language phrases, but the details of their outcomes may change. For example, a family may include

**Valid input:**
Two strings of at most 100 characters, comprising alphanumeric characters (A-Z and 0-9), punctuation, and whitespace.

**Example inputs:**
"one", "two"
"first argument", "second argument"
"", "1,000,000 arguments, with a cup of tea"

```python
def interleave(a, b):
  args = [a, b]
  max_length = max([len(i) for i in args])
  r = []

  for j in range(max_length):
    for k in args:
      try:
        r.append(k[j])
      except:
        pass

  return "".join(r)
```

```python
def interleave(a, b):
  args = [a,b]
  result = []
  m = max(map(len, args))
  for i in range(m):
    for arr in args:
      try:
        result.append(arr[i])
      except:
        pass
  return "".join(result)
```

```java
public static String ANON(String l1,
                          String l2){
  ArrayList<Character> c =
    new ArrayList<>();

  for (int i = 0; i < l1.length(); i++){
    c.add(l1.charAt(i));
    c.add(l2.charAt(i));
  }

  return String.valueOf(c.toArray());
}
```

```python
def ANON(a, b):
  arg = [a,b]
  r = []
  m = max(map(len, arg))
  for i in range(m):
    for arr in arg:
      try:
        r.append(arr[i])
      except:
        pass
  return "".join(r)
```

Fig. 1. Similar and dissimilar snippet pairs in the same task family with input domains and examples. The left pair is classified [*clones*, *same* language, *similar* structure, *meaningful* names]. The right pair is classified [*non-clones*, *different* languages, *dissimilar* structures, *obfuscated* names].

algorithms that order a list of objects without specifying on what feature to order them.

A task consists of a side-by-side pair snippets from the same family, never between families. Our study environment features this pair exclusively, rather than working in real-world interfaces where code occurs alongside competing signals like crowdsourced quality signals, natural language discussion, or other dissimilar code. Additionally, we defined a domain and examples of expected input for each family, as in Figure 1, to avoid issues about the limits of types on specific architectures.

### B. Artifact Collection

To improve the naturalness of our tasks, we curated existing snippets from the web rather than designing new ones. We set goals for snippet families: snippets should be compact and single functions (e.g. $\leq 25$ lines), have nontrivial or branching logic with no syntactic errors, and rely on built-in features or standard libraries. To find candidates, we explored code clone benchmarks [18], [19], [20], [21], [22], code search engines, open-source platforms like GitHub, and practice websites like Codewars[1] and LeetCode.[2] We manually modified some snippets to control our variables but wrote none from scratch.

We identified the 6 task families in Table I throughout the aforementioned resources. Treating every independent variable as binary, we needed $2^4 = 16$ snippet pairs for full coverage. For behavior, we manually analyzed snippets and made tests from edge cases and 10,000 randomly generated inputs to establish a heuristic for our ground truth of behavior similarity. For language, we focus on Java and Python because of their popularity, also meaning that any snippets in Python should be valid for both Python 2 and Python 3. For structure, our use of cross-language pairs meant we could not use simple token similarity as our measure. Instead, the first author manually classified snippets in relation to algorithmic design choices identified in the *Structural Variation* column of Table I. For naming, we ensured the function name represented the task and otherwise preserved the variable names in the body as originally found, even if they were abbreviated. To obfuscate names, we replaced function names with `ANON` and reduced variables names to at most four characters. Figure 1 demonstrates our method: the right-side method in each pair is the same snippet, except that the left instance has meaningful names and the right is obfuscated.

From the first three variables and six families, we created 48 snippet pairs from 47 unique snippets reused up to three times. This doubled to 96 pairs overall after obfuscating names. However, after study execution, we reclassified the structures of six of the 48 distinct snippet pairs to refine structural classifications, so the treatments were not perfectly balanced in the analysis. Furthermore, in case anyone volunteered for both interview and survey, we also curated 3 partial snippets pairs as back-ups: algorithms for factoring an integer, generating string permutations, and checking whether two arrays are circularly identical. These snippets, along with all the other study data, are included on our Zenodo repository [23].

### C. Think-Aloud Tasks & Interviews

*1) Environment:* Due to pandemic conditions, we used Zoom to host, record, and transcribe a draft of the conversation, which the first author (henceforth the interviewer) refined. We conducted the interview with a screen-shared slideshow for all visual content and coordinated it with oral explanation.

*2) Protocol:* We designed interviews to be 30 minutes. Every interview had three phases: concept introductions, tasks, and a reflective interview. For concepts, we defined behavioral similarity and showed examples of clone and non-clone pairs. We then explained the think-aloud protocol, specifying our interest in not just conclusions but in impressions, assumptions, and continuous focus throughout the task. We also clarified what other actions they could take, such as consulting API documentation but not rewriting and executing the algorithm. For tasks, we drew two or three pairs, depending on time taken, from the first four tasks in Table I, as well as the three partial families in Section III-B because one participant also participated in the survey. When a participant concluded their investigation with a yes or no answer, the interviewer asked them to reflect on their strategy. After the final pair, we asked

[1]https://codewars.com/
[2]https://leetcode.com/

TABLE I
EXPERIMENTAL TASKS WITH MULTIPLE VARIATIONS. ALL STRING INPUTS WERE CAPPED WITH AT MOST 100 CHARACTER LENGTH.

| | Task Family | Behavioral Variation (IV-1) | Structural Variation (IV-3) | Input Domain |
|---|---|---|---|---|
| 1 | Bubblesort | Sort an array of integers.<br>• In ascending order<br>• In descending order | Does it manage looping with:<br>• nested for-loops<br>• one for-loop and a while-loop | one integer array: $\leq$1M items between $\pm$1B |
| 2 | CamelCaser | Convert snake_case to CamelCase.<br>• Capitalize first character<br>• Ignore first character | Does it<br>• iterate over characters<br>• split by underscore and iterate over words | one string: alphanumeric and underscores |
| 3 | Interleave | Combine letters from strings<br>• Append excess characters<br>• Omit excess characters<br>• Throw exception | Is it:<br>• a single-loop solution for two strings<br>• a double for-loop for indefinite strings<br>• while+for loop solution for indefinite strings | two strings: alphanumeric, punctuation, and spaces |
| 4 | Deduplicate | Deduplicate adjacent characters<br>• Replace all duplicates ('aaaa' $\rightarrow$ 'a')<br>• Replace pairs of two ('aaaa' $\rightarrow$ 'aa')<br>• Remove pairs of two ('aaaa' $\rightarrow$ '') | Does it:<br>• iterate over adjacent characters<br>• store the last unique character | one string: alphanumeric, punctuation, underscores and spaces |
| 5 | Anagram (Survey Only) | True if two strings are exact anagrams<br>• Case sensitive<br>• Case insensitive | Does it:<br>• compare sorted strings<br>• count character frequency? | two strings: alphanumeric ands spaces |
| 6 | Balanced (Survey Only) | True if all brackets are closed, in order<br>• Match exact bracket type<br>• Ignore bracket type<br>• Allow unclosed | Does it:<br>• build a hash-map of bracket characters<br>• use an if-else sequence or switch statements? | string: characters in "{}[]()" |

them on their strategies overall and where they employ these strategies in practical software development contexts.

*3) Recruitment:* We distributed our study and consent information at software companies, social media like Twitter and LinkedIn, and university mailing lists. Undergraduate participants received extra credit in a university course for participation, but we did not offer further compensation.

*4) Population:* We had 16 total participants, as listed in Table II: four undergraduate students (U1 – U4), five graduate students (G1 – G5), and seven professionals (P1 – P7). Their years of programming experience ranged from 2 to 20, as shown in the *Years Exp.* column.

*5) Analysis:* To answer RQ1 and RQ2, we analyzed the think-aloud scripts from the interviews for how the participant's narration focused on and moved between the snippets in the pair. Additionally, the interviewer used open card sorting to organize statements to common questions by primary themes. To partially address RQ3, the tasks were analyzed in terms of correctness with our ground truth, although we did not fit these data into a statistical model because of our low sample size compared to number of factors. Table II also shows all tasks, treatments, and outcomes. For example, participant U1 had three tasks, including a pair of Java snippets (J) with similar structures (S) and meaningful names (N) which were indeed clones (C), yielding the shorthand (**J S N C**). In this case, U1 labeled them as non-clones, but this was incorrect (✖) by our definition.

### D. Survey

*1) Environment:* We designed the survey to be 45 minutes on Qualtrics,[3] a survey platform that can randomize task order and balance treatments. The survey contains four sections: screening questions, conceptual introductions, tasks, and exit questions. The screening questionnaire comprised the consent form, self-assessments of Java and Python experience, and two questions about snippets in each language to test comprehension. Participants continued if they reported some experience in both languages and got at least two code questions correct.

The conceptual introduction defined behavioral code clones as "*programs that give the same output for all of a given range of inputs, even if they don't follow the same algorithm*," and included interactive examples. A task includes an image of a randomly selected pair of snippets from one of the families in Table I and as many as four questions: are they clones for the valid domain and for an unrestricted domain, which inputs demonstrate difference in output, and what were the most significant cues. Each participant completed an initial set of four tasks in a random order and an attentional task to filter out low-quality responses, and they had the option to complete two more experimental tasks. At the end, participants optionally reflected on their overall strategies.

*2) Recruitment:* We piloted the study with eight volunteers from a graduate school and a software engineering conference. After refining the study, we then reached out to populations available to us in discussion boards (CS/SE boards on Reddit, Gitter), university student mailing lists, social media (Twitter and LinkedIn), personal connections, and Mechanical Turk.

[3]https://www.qualtrics.com/

The first 100 participants were awarded a $10 Amazon gift card, and all participants who submitted emails were entered into a raffle for a $50 dollar gift card.

*3) Population:* Altogether, 288 people attempted the qualifying questionnaire and 124 qualified. Overall, we excluded 29 participants and their 161 observations for quality issues and retained 95 participants and 461 observations from the experimental task. Of the participants, 68 were graduate students, 17 were current software professionals, and the rest held other positions around academia or self-employment.

*4) Analysis:* To address RQ3, we used a mixed-effects logistic regression model to fit our binary response variable to our independent features and control for repeated observations from the same participant and task families. In the vocabulary of the R package *glmer* for mixed-effects models, the formula (interactions omitted to save space) is the following:

$$\begin{aligned} correct \sim\ & language\ +\ structure\ +\ naming \\ & +\ clone\_truth\ +(language * structure)\ +\ ... \\ & +\ (1|participant)\ +\ (1|task) \end{aligned}$$

With 461 observations, we pruned higher-level and insignificant interactions to converge at the model in Table IV.

## IV. Findings

In this section, we present data for our research questions about *when*, *by what strategies*, and *how accurately* developers compare code.

### A. RQ1: When do developers compare code?

Student participants (U4, G1, G3) discussed *learning* contexts: a developer compares programs to discover, evaluate, and retain techniques to a well-defined problem to improve personal competency. This can happen in structured lessons (e.g. U4 learning how to analyze computational complexity) or as voluntary participation in programming practice communities (e.g. LeetCode for G1, on Discord for G3). Participants may first prepare their own algorithm to solve a problem and compare it to how other developers solved it.

Among professional developers (P2, P5, P6), *review* also employs code comparison: the developer compares code to verify the maintenance or evolution of properties between versions to improve a software system. P5 illustrates this case in the question, "*does the evolved version do the same thing when it is supposed to and do the different thing when it's supposed to.*" Developers in this context often have project-specific experience that contextualizes code in the review, along with documentation and testing tools.

*Refactoring* (P2, P4, P5, P7) and *repair* (P1, P4, P5, P7) are two activities that can contextualize reviews and constrain comparative goals. For refactoring, developers compare code to verify nonfunctional changes while preserving functional behavior. P7 details the many judgments in the refactoring mindset, considering "*why the simpler, more beautiful, more elegant, more efficient version doesn't actually produce the same answer, [and] whether it's a better answer, or worse answer.*" In repair, developers compare code to isolate faulty functional properties in behavior. P1 describes their experience: "*Frequently [the code versions] look very similar, but there will be some small change that I have to pick up on.*" Both require comparing code versions to verify evolution.

However, when asked about where they compared code, our participants also discussed situations where they did not have two concrete alternatives of code (as in our tasks), but instead an abstract goal and an instance of concrete code as a tentative reference point. For example, while *constructing new code from a model*, developers implement new features by using existing code as an example of what they want to write without directly copying it (U1, U3, G2, G3, G4, P3):

> (P3) I'm trying to do a very similar thing, but there are subtle differences...I use what I have as a guide but I'm constantly comparing it to what I'm writing to make small changes to fit my problem.

We also discussed Q&A websites like Stack Overflow or code search engines like that on GitHub with several participants. Although multiple snippets co-exist in the same interfaces, participants detail an opportunistic strategy which prefers surface signals rather than a deep comparison of options:

> (G5) I first check for the green checkmark [on Stack Overflow]. Okay, this is a good answer and then, if it doesn't work, then just moving on to the next one and see how that works. But not a direct comparison for each one.

This is more akin to recognizing satisfactory snippets by mentally held requirements than it is to concrete code features.

> RQ1: Code comparison supports learning contexts by clarifying the impact of different designs and review contexts by verifying code evolution. In example-driven construction and search contexts, developers also consider indirect comparisons with requirements instead of code.

### B. RQ2: With what strategies do developers compare code?

By definition, comparative comprehension over code involves two types of movement for the developer's focus: first, *between features within a single snippet* to form a behavioral understanding; second, *between snippets* to estimate similarity. On this foundation, we elaborate other phenomena that characterized the interpretation of difference.

**The shape of code hints at challenge and influences the starting point.** Participants commented on both extreme visual similarity (G4: *"This looks like two people tried to copy each other's code"*) and extreme difference (P2: *"It's three times as long as the one on the right."*). Starting a task, participants most often refer to the left snippet alone or similar elements from both, but in four out of 45 instances, participants focused on the right snippet first when some property made it more accommodating. For example, participants G2, P5 and P7 saw the same pair of camel-casing algorithms (naming aside), but whereas G2 and P7 began with the right snippet because of it was shorter than the left snippet, P5 began on the left because they disliked the list comprehension in the right. Another reason for choosing a particular starting point was in the preference for a particular language over the other.

TABLE II
ALL INTERVIEW TASKS WITH CORRECTNESS. WITHIN EACH PARTICIPANT, TASK ORDER WAS RANDOM.
*: A PARTIAL TASK DESCRIBED IN SECTION III-B.

| | Position | Years Exp. | Bubblesort | CamelCase | Interleave | Duplicates | Circular* | Permute* | Factors* |
|---|---|---|---|---|---|---|---|---|---|
| U1 | Undergraduate | 2- 5 | J S N C ✖ | | x s n c ✖ | | | P S N C ● | |
| U2 | Undergraduate | 2- 5 | | J S n c ● | x s N C ● | x S N C ● | | | |
| U3 | Undergraduate | 2- 5 | x S n C ● | J S N C ● | x S n C ● | | | | |
| U4 | Undergraduate | 2- 5 | | J S N c ● | | x S n C ● | | | |
| G1 | Graduate | 2- 5 | P s n C ● | | P S N c ✖ | | | | |
| G2 | Graduate | 5-10 | x S n C ✖ | P s N c ○ | | | | | |
| G3 | Graduate | 2- 5 | J s n c ● | P s n c ● | x s N c ● | | | | |
| G4 | Graduate | 5-10 | J S N C ● | J S n C ✖ | | x s N c ○ | | | |
| G5 | Graduate | 2- 5 | | | | | P s N C ✖ | P S n C ● | x s N c ● |
| P1 | Professional | 2- 5 | P s N C ✖ | P S n C ✖ | P S n c ○ | | | | |
| P2 | Professional | 5-10 | x S N C ● | J S n C ● | x s N c ✖ | | | | |
| P3 | Professional | 5-10 | J S n C ● | J S N C ● | | x s n c ✖ | | | |
| P4 | Professional | 2- 5 | J s N c ● | P S N C ● | x s n c ● | | | | |
| P5 | Professional | ≥20 | P s n c ? | P s n c ○ | P S N C ● | | | | |
| P6 | Professional | 10-20 | | | x s N C ● | | P s n C ● | | x s n c ● |
| P7 | Pro., non-SE | ≥20 | P s N c ✖ | P s N c ● | P S n C ● | | | | |

Format: {*Language Structure Names Clone-Truth Correctness*}

| | | | |
|---|---|---|---|
| *Language* | [**J**ava \| **P**ython \| x: crossed] | *Names* | [**N**: meaningful \| n: obfuscated] |
| *Structure* | [**S**: similar \| s: dissimilar] | *Clone-Truth* | [**C**: clones \| c: non-clones] |
| *Correctness* | [●: correct \| ○ : correct for wrong reasons \| ✖: incorrect \| ?: unresolved] | | |

**Meaningful names inform expectations which serve as a foundation for comprehension and comparison.** Names can transform the initial task from "What do these do?" to "How does this meet my expectations?" P1 explains, "*When I saw it was called bubblesort I immediately knew what to expect, versus when they're ANON and I don't know what to expect.*" Developers may nevertheless recognize obfuscated code as a familiar schema, and this realization may recontextualize the comparison. In one instance of obfuscated bubblesort algorithms, participant G1 initially concluded that two algorithms were non-clones because one defined a loop variable without using its value, whereas the other used all variables. However, further investigation triggered their recollection of bubblesort algorithms, and G1 reevaluated the other snippet's variables not in terms of value but in terms of why the loop had to iterate even if it did not use the indexing variable in calculations.

**Developers may delay moving between snippets to comprehend one entire snippet first.** A major decision of a comparison is when to switch focus. However, some participants established a foundation by comprehending the first snippet exclusively, at least broadly, whereas others cross repeatedly. In our 45 observations, 29 tasks began with narrations of an exclusive focus on one snippet, whereas the other 16 mixed elements from both before reaching the end of either.

**Developers move between snippets to map and evaluate similar features.** From our analysis of participants' narrated thoughts, we note three general strategies for alternating focus: *textual, structural, and schematic comparison*.

1) In **textual comparison**, developers rapidly switch between characters or tokens of highly similar pairs. Developers can eschew the effort of comprehending identical statements, as P4 demonstrates with similar camelcase algorithms: "*since it was identical I didn't even bother trying to figure out what this was doing.*" If variable names differ, developers can maintain a mental map of corresponding variables.
2) In **structural comparison**, developers switch focus often to map structural features. When the developer comprehends a chunk of tokens as discrete behavior, they search the other snippet for a corresponding chunk. If no matching feature is found, the developer must test the significance of the difference.
3) In **schematic comparison** (as in the "schema" of an algorithm), developers switch infrequently, understanding algorithms holistically rather than as mappable features. In pairs of extreme difference—no overlapping structural features or different paradigms—this is the primary option.

Furthermore, the type of comparison can be influenced in response to similarities in the code. For example, G3 narrated moving between schematic and textual styles:

> (G3) Those [loops] don't look similar at all, so I can't compare them directly, I have to compare them with a mental model. Whereas the two if-statements I was like, 'Oh hey, these are almost identical, they are only a little bit different. What's the difference?"'

Likewise, if the developer discovers code chunks that may be analogues through structural comparison, they may narrow their focus to those program parts in a style of smaller-scoped schematic comparison.

**Developers recognize that multiple differences can compensate for each other to create similar behavior overall.** For example, when U2 structurally compared character-deduplication algorithms, they note that the Java definition `String result = "" + orig.charAt(0);` includes behavior that the corresponding Python definition `res = ""` does not. However, separate differences in the return statements compensate for initial differences: Java's `return result;` vs Python's `return orig[0] + res`. Otherwise, the difference may be insurmountable, and the de-

TABLE III
NUMBER OF CORRECT SURVEY ANSWERS (✓) PER RESPONSES IN TREATMENT ($Qs$)

| IV-2 Language | IV-3 Structure | IV-4 Names | Clones Qs | Clones ✓ | Clones % | Nonclones Qs | Nonclones ✓ | Nonclones % | All Qs | All ✓ | All % |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Same | Similar | Meaningful | 24 | 22 | 0.92 | 23 | 12 | 0.52 | 47 | 34 | 0.72 |
| | | Obfuscated | 35 | 32 | 0.91 | 15 | 9 | 0.60 | 50 | 41 | 0.82 |
| | Dissimilar | Meaningful | 28 | 25 | 0.89 | 33 | 15 | 0.45 | 61 | 40 | 0.66 |
| | | Obfuscated | 23 | 19 | 0.83 | 36 | 15 | 0.42 | 59 | 34 | 0.58 |
| Different | Similar | Meaningful | 33 | 32 | 0.97 | 25 | 9 | 0.36 | 58 | 41 | 0.71 |
| | | Obfuscated | 26 | 20 | 0.77 | 28 | 14 | 0.50 | 54 | 34 | 0.63 |
| | Dissimilar | Meaningful | 35 | 29 | 0.83 | 34 | 13 | 0.38 | 69 | 42 | 0.61 |
| | | Obfuscated | 28 | 20 | 0.71 | 35 | 12 | 0.34 | 63 | 32 | 0.51 |
| | | Total | 232 | 199 | 0.86 | 229 | 99 | 0.43 | 461 | 298 | 0.65 |

TABLE IV
EFFECTS IN LOGISTIC REGRESSION MODEL PREDICTING CORRECT RESPONSES. 461 OBSERVATIONS FROM 95 PARTICIPANTS AND 6 TASKS.

An odds ratio of 1.8 for similar structures means that the odds a participant will respond correctly given similarly structured code is estimated to be 1.8 times the odds of a correct answer given dissimilar structures, all other factors held constant.

| **Fixed Effects** | **Odds Ratio** | **Est.** | **Std Err** | **p-value** | |
|---|---|---|---|---|---|
| (Intercept) | | -0.719 | 0.288 | 0.013 | * |
| True Clones | 5.833 | 1.764 | 0.343 | <0.001 | *** |
| Similar Structures | 1.829 | 0.604 | 0.251 | 0.016 | * |
| Same Languages | 1.665 | 0.510 | 0.242 | 0.035 | * |
| Meaningful Names | 0.833 | -0.183 | 0.302 | 0.545 | |
| *Interactions* | | | | | |
| Clones:Names | 3.016 | 1.104 | 0.521 | 0.034 | * |
| **Random Effects** | | **Variance** | | | |
| Participant | | 0.361 | | | |
| Task | | 0.076 | | | |

*: $p < .05$, **: $p < .01$, ***: $p < .001$

veloper is confident that behavior diverges even if a exhaustive understanding of the code has not been reached.

> RQ2: When viewing code alternatives, developers attend to initial cues of similarity like size and complexity. They can, but do not have to, build an initial foundation by reading names or a full snippet in isolation before comparison. In response, they can perform comparison at multiple layers—at text, structure, or the schema—in order to pinpoint differences and evaluate their significance.

### C. RQ3: What factors impact accuracy?

*1) Think-Aloud:* As in Table II, 33 of 45 tasks were answered correctly (73.3%), 11 incorrectly (24.4%), and one was unresolved because the participant felt they lacked information. However, in four of the correct answers, participants narrated erroneous reasoning that aligned with the correct binary choice. Counting these cases and the unresolved task as incorrect answers, the rate of correctness is 29 of 45 (64.4%). We grouped them 16 wrong answers by the apparent misunderstandings that explain their inaccuracy:

- *Control flow misunderstanding* (6): The participant inaccurately predicts which branch a program will take or how often it takes it. *Example*: U1 misinterprets `i <= length - 2` and `i < length - 1` as different loop limits.
- *API misunderstanding* (2): The participant inaccurately anticipates a function call's effect. *Example*: P1 expects python `range(x)` to iterate from 1 to x instead of 0 to x.
- *Data misunderstanding* (2): The participant has an inaccurate belief about a variable's value. *Example*: G5 mistakes a one-space string literal for an empty literal.
- *Multiple misunderstandings* (2): The error comes from multiple misconceptions. *Example*: G4 misinterprets what the Java `substring(x)` method does and misinterprets how many times it is called in a loop.
- *Overlooked feature differences* (2): The participant anticipates equivalent behavior and never mentions important structural differences. *Example*: G1 misses that two algorithms substitute `min` and `max` for string length calculations.
- *Overlooked input case* (2): The participant does not consider an input that would demonstrate difference. *Example*: P3 correctly identifies that two snippets are clones when given sequences of 2 letters but does not consider sequences of 3+.

Of these errors, the misunderstandings stymie the comprehension of individual snippets and therefore stymie comparison as consequence. Overlooking features or input cases, however, deal directly with how developers look for distinctions.

*2) Survey:* Table III displays the proportion of correct answers for each treatment. For example, when the snippets were *clones*, the language was the *same*, the structures *similar*, and the names *meaningful*, of the 24 responses (*Qs*), 22 (92%) were correct (✓). For *nonclones* with the same treatment otherwise, 12 responses out of 23 (52.2%) are correct. The lowest correctness rate occurs when all features are dissimilar or obfuscated. The overall correctness was 64.6%, but with a substantial difference between when the algorithmic task had true clones (85.8%) versus when they did not (43.2%).

In Table IV, five terms of our model in Section III-D4 have significant influence ($\alpha = .05$) on the outcome: the intercept, clone truth, structural similarity, shared languages, and the interaction between clone truth and clear names. Table IV also shows the variance that our model estimated for each random effect: variance in participant differences (.361) is more than in task differences (.076). Clone truth has the highest odds ratio:

the odds a participant answers correctly changes by a factor estimated to be 5.836 between non-clones and clones when all other factors are held constant. Although the inclusion of meaningful names did not itself have a significant effect, the clone-name interaction did. Specifically:

- For non-clones, participants are correct 43% of the time with meaningful names versus 44% when obfuscated.
- For clones, participants are correct 90% of the time with meaningful names and 81% when obfuscated.

RQ3: In think-aloud tasks, participants accurately compared snippets in 73% of tasks, although at times with incorrect reasoning. The most common errors affected predicting control flow. In survey tasks, participants were correct in 65% of tasks, but significantly less correct for non-clones than clones. Answers were more often correct when snippets were similar and more incorrect when dissimilar. Clone truth, structure, and language significant impacted participants' correctness, and names interacted with clone truth.

## V. Discussion

In this section, we interpret results across research questions and generalize their implications for industry and research.

### A. Impact of Design and Context

Table IV shows that differences in the algorithms' syntactic appearance—which structure and language affect—interfere with the ability to judge behavior. However, the largest effect on correctness comes from the ground truth of behavior. This result raises questions because participants did not have direct access to this factor, and since our correctness was binary, it suggests that our participants overestimate how many pairs are behaviorally equivalent. One interpretation is that the assumptions of our environment incline developer towards clones as a default hypothesis. Comprehending code is time-consuming, and in non-experimental contexts, developers have to explore many programs that do not stand as alternatives for their case [24]. Because of this effort, developers are often satisfied with "close-enough" rather than perfect fit [25], [8]. However, the question in our case was not close-enough but equivalence, and given that our snippets were curated by virtue of their existing similarities, participants may have terminated investigation there rather than continue exhaustively.

Future work should explore how assumptions of similarity impact comparative comprehension, especially when alternatives are not as neatly extracted from their original context as here. Additionally, future models of comparative comprehension should consider additional sources of variation, such as developer experience, which has been shown to influence individual comprehension [26]. We can also improve the current factors, such as algorithmic structure, which our binary classification treats broadly. A more quantitative approach like cross-language AST similarity [27], or the decomposition of the structure into small features, could refine results.

### B. Comparison's Relationship to Comprehension

Although superficial comparison can start an investigation, behavioral comparison builds on code comprehension of individual snippets, although not necessarily total comprehension. For participants to understand behavior, our results highlight strategies that correspond to bottom-up and top-down investigation from comprehension literature [28]. For examples of top-down comprehension, meaningful names establish expectations which change the question of investigation, and understanding one algorithm in full can also influence the expectations for the other. However, given no cues to code behavior before the task, participants also had to construct programs from small features from the bottom up as well.

However, comprehension strategies in this experiment also included making associations between analogical features in snippets. This activity is similar to the process of *alignment* within pedagogical literature, where learners draw associations between common features of examples of a principle [29]. In our case, individual comprehension could be supplemented or even interrupted by a sudden switch between snippets to search out whether a feature was shared and to create that mapping or association if so. If not, that could inform a conclusion of non-clones, but developers must also consider whether the differences could accumulate into the same behavior once fully understood and translated. In this way, even when snippets are in the same language, there are concerns of knowing how to translate between representations, which is an effortful and error-prone process [15]. Tools that clarify transfer issues could avoid misconceptions and reduce effort.

### C. Implications for Research

Comparative comprehension resonates with challenges in program verification, validation, and dynamic differentiation. For example, one goal of testing is to catch regression errors in evolving programs, where behavior changes when the developer intended to preserve it [30]. Our results may emphasize the issues in comparing over similar texts that make these bugs difficult to detect manually. Furthermore, efficiently finding inputs which demonstrate difference is a topic of interest in program synthesis literature [31]. Tool that generate differentiating input could address challenges in comparison by supporting the developer with concrete cases, thus saving them the search and reasoning work.

Future work should also extend this research to facilitate multiple dimensions of comparison. While input-output behavior is essential to describing what a program does, it does not represent all that developers care about. Runtime complexity and memory consumption distinguishes algorithms even when behavior is equivalent, and the impact of reusable code alternatives on the maintainability or reusability of programs can be difficult to anticipate [32].

Lastly, if comparison works by connecting similarities in and beneath text, future studies should attend to the physical arrangement of code. As alignment is important for case comparison in pedagogy [33], styling code to physically align

analogical parts may highlight implicit similarities. Experimental interfaces like Bragdon and colleagues' Code Bubbles, suggest that proximity helps comprehension beyond simply reducing navigation [34].

## VI. Threats

*a) Construct threats:* We model variation in four factors but more may exist that influence comparative comprehension. For example, we measured structure through qualitative assessment which does not capture every design choice. There are also different ways to obfuscate names to remove meaning.

*b) Internal threats:* We may have misclassified observations in our qualitative protocols or classifications in our quantitative protocols since we used a single-coder approach. Additionally, think-aloud protocols capture the participant's approximation of their reason but not the full set of influences and distraction. The survey methodology removes the pressure of observation of think-aloud tasks but also removes our ability to clarify misunderstandings and observe attention.

*c) External threats:* Our survey disproportionately attracted graduate students on the mailing list, skewing our representation from populations estimated in other sources [35]. We had fewer than 10 programmatic tasks; this does not totally cover the range of snippets developers encounter in the wild. Lastly, our choice of programming languages represents two very popular language with similar capabilities but it does not fully represent paradigmatic diversity.

## VII. Related Work

In this section, we summarize research in comprehension, comparison, and clones that is the foundation to our work.

*a) Code Comprehension:* Numerous theories model how developers bridge text and behavior. For example, von Mayrhauser and Vans combine top-down strategies, which apply when the developer is verifying expectations, and bottom-up strategies, which apply in unfamiliar territory [28]. Soloway and colleagues' experiment with typical and atypical programs suggests that developers can draw from repertoires of programming plans, or mental representations of the elements and order of a program [26]. Additionally, many studies focus on the influence of specific textual features on comprehensibility, such as identifier style [36], [17] or structural complexity [37], [38], although Scalabrino and colleagues suggest that many current metrics are insufficient for drawing clear conclusions [10]. We draw on studies like these to set a foundation for program comprehension, but our focus on structurally unrelated snippets complicates models in which developers looking for a unified picture of a single system.

*b) Code Comparison:* Two practical contexts of comparison are education and reuse. In education, Alfieri and colleagues' meta-analysis of science and math pedagogical literature confirms that case comparison—showing multiple examples of a principle and supporting students to discover their commonalities—is an effective teaching strategy, and they deconstruct the activity into a pedagogical model [29]. Patitsas and colleagues demonstrated its application to computer science education: students who study two algorithms side-by-side had greater increases in procedural knowledge and flexibility compared to students who study them sequentially [33]. Although these studies focus on classrooms, their evidence motivates our case of comparison more broadly.

In the context of reuse, Détienne frames reusing code in terms of mapping between known approaches and abstract needs [39]. Other directions focus on navigating similarity amid larger bodies of code. Working from Information Foraging Theory (IFT) [40], research by Ragavan and colleagues observes developers in a repository of similar versions [24], as Kuttal and colleagues do for end-user programmers [41]. They demonstrate that the awareness of code similarity better predicts how developers forage [12]. Though this research explores comparison too, it does so in the context of navigation, whereas we focus more on local comparison strategies.

Because of applications in review and reuse, some research supports comparison with tools. Many algorithms exist to analyze and highlight differences in text [42] or parse trees [43]. Some tools, such as Cottrell and colleagues' GUIDO [44] or Glassman and colleagues' EXAMPLORE [45], build interfaces to explore varied examples, and other tools record edit events in the IDE to explain differences in program evolution [46], [47]. Our empirical data, which shows that comparison is nontrivial, motivate advancements on tool design such as these.

*c) Code Cloning:* We refer to significantly similar code excerpts as "clones," and the literature outlines several types of clones: clones types 1 through 3 refer to a spectrum of syntactically similar code, and Type-4 clones, or semantic clones, perform the same operation [9]. Manual verification of clones is nontrivial, as Walenstein and colleagues [48], [49] as well as Kasper and colleagues [50] both uncover issues of experts disagreeing on definitions of clones. Furthermore, Charpentier and colleagues' experiment discovered not only that it was difficult for judges to disagree with each other but also themselves over time [51]. Our research complements the challenge of clone recognition: even within a settled definition of clone, there are inherent difficulties to accurately understanding the behavioral differences from its text.

## VIII. Conclusion

Comparison is relevant as long as there are more than one way to solve a problem. With that in mind, we ran an experiment which presented developers with code snippets pairs and asked them to identify whether their behaviors were equivalent, controlling for their behavior, language, structure, and names. Through our interviews and surveys, we captured how developers' comparison strategies allow them to piece together behavior and align the ways two algorithms perform. However, participants overestimated behavioral equivalence, and design differences between snippets interfere with the ability to clarify the underlying behavioral differences. As we devise new ways to navigate code alternatives, we should weigh the barriers to comprehension that interfere with developers choosing within the sea of options.

## IX. Acknowledgments

This work is funded in part by NSF SHF #2006947, #1749936, and #1714699. Thank you to Emily Griffith for statistical advice, to all reviewers, Kai Presler-Marshall, Gina Bai, Nischal Shrestha, Souti Chattopadhyay, and Omar Sheikh for their writing feedback, to George Mathew for reusable code, and to all study participants for their cooperation.

## References

[1] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[2] V. Käfer, S. Wagner, and R. Koschke, "Are there functionally similar code clones in practice?" in *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. IEEE, 2018, pp. 2–8.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293.

[4] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[5] J. Siegmund, "Program comprehension: Past, present, and future," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 13–20.

[6] K. Fisler, "The recurring rainfall problem," in *Proceedings of the tenth annual conference on International computing education research*, 2014, pp. 35–42.

[7] A. Blass, N. Dershowitz, and Y. Gurevich, "When are two algorithms the same?" *The Bulletin of Symbolic Logic*, pp. 145–168, 2009.

[8] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–37, 2014.

[9] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[10] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?" in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 417–427.

[11] M. J. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," in *11th working conference on reverse engineering*. IEEE, 2004, pp. 70–79.

[12] S. S. Ragavan, B. Pandya, D. Piorkowski, C. Hill, S. K. Kuttal, A. Sarma, and M. Burnett, "Pfis-v: modeling foraging behavior in the presence of variants," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 6232–6244.

[13] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 286–28 610.

[14] P. Mayer and A. Bauer, "An empirical analysis of the utilization of multiple programming languages in open source projects," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 2015, pp. 1–10.

[15] N. Shrestha, T. Barik, and C. Parnin, "It's like python but: Towards supporting transfer of programming language knowledge," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 177–185.

[16] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 2006, pp. 3–12.

[17] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 217–227.

[18] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[19] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[20] F. Al-Omari, C. K. Roy, and T. Chen, "Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge," in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 2020, pp. 57–63.

[21] M. Kamp, P. Kreutzer, and M. Philippsen, "Sesame: a data set of semantically similar java methods," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 529–533.

[22] G. Mathew, C. Parnin, and K. T. Stolee, "Slacc: Simion-based language agnostic code clones," *arXiv preprint arXiv:2002.03039*, 2020.

[23] *Understanding Similar Code through Comparative Comprehension*. Zenodo, Apr. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6419729

[24] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, "Foraging among an overabundance of similar variants," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 3509–3521.

[25] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–44, 2013.

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

[27] G. Mathew and K. T. Stolee, "Cross-language code search using static and dynamic analyses," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 205–217.

[28] A. von Mayrhauser and A. M. Vans, "Program understanding behavior during debugging of large scale software," in *Papers presented at the seventh workshop on Empirical studies of programmers*, 1997, pp. 157–179.

[29] L. Alfieri, T. J. Nokes-Malach, and C. D. Schunn, "Learning through case comparisons: A meta-analytic review," *Educational Psychologist*, vol. 48, no. 2, pp. 87–113, 2013.

[30] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 397–406.

[31] A. Solar-Lezama and R. Bodik, *Program synthesis by sketching*. Citeseer, 2008.

[32] S. N. Woodfield, D. W. Embley, and D. T. Scott, "Can programmers reuse software?" *IEEE Software*, vol. 4, no. 4, p. 52, 1987.

[33] E. Patitsas, M. Craig, and S. Easterbrook, "Comparing and contrasting different algorithms leads to increased student learning," in *Proceedings of the ninth annual international ACM conference on International computing education research*, 2013, pp. 145–152.

[34] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr, "Code bubbles: a working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 2503–2512.

[35] Stack Overflow Blog, "Developer survey results," *Retrieved August*, 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019\#overview

[36] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Software Engineering*, vol. 18, no. 2, pp. 219–276, 2013.

[37] A. Mohan, N. Gold, and P. Layzell, "An initial approach to assessing program comprehensibility using spatial complexity, number of concepts and typographical style," in *11th Working Conference on Reverse Engineering*. IEEE, 2004, pp. 246–255.

[38] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms—what really affects code complexity?" *Empirical Software Engineering*, vol. 24, no. 1, pp. 287–328, 2019.

[39] F. Détienne, "Reasoning from a schema and from an analog in software code reuse," *arXiv preprint cs/0701200*, 2007.

[40] P. Pirolli and S. Card, "Information foraging in information access environments," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1995, pp. 51–58.

[41] S. K. Kuttal, S. Y. Kim, C. Martos, and A. Bejarano, "How end-user programmers forage in online repositories? an information foraging perspective," *Journal of Computer Languages*, vol. 62, p. 101010, 2021.
[42] J. W. Hunt and M. D. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.
[43] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.
[44] R. Cottrell, B. Goyette, R. Holmes, R. J. Walker, and J. Denzinger, "Compare and contrast: Visual exploration of source code examples," in *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2009, pp. 29–32.
[45] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing api usage examples at scale," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.
[46] F. Jacob, D. Hou, and P. Jablonski, "Actively comparing clones inside the code editor," in *Proceedings of the 4th International Workshop on Software Clones*, 2010, pp. 9–16.
[47] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: generating concise linked code differences," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 679–690.
[48] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems creating task-relevant clone detection reference data." in *WCRE*, vol. 3. Citeseer, 2003, p. 285.
[49] A. Walenstein, "Code clones: Reconsidering terminology," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
[50] C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. Van Rysselberghe, and P. Weißgerber, "Subjectivity in clone judgment: Can we ever agree?" in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
[51] A. Charpentier, J.-R. Falleri, F. Morandat, E. B. H. Yahia, and L. Réveillere, "Raters' reliability in clone benchmarks construction," *Empirical Software Engineering*, vol. 22, no. 1, pp. 235–258, 2017.