Thinking Aloud about Confusing Code

A Qualitative Investigation of Program Comprehension and Atoms of Confusion

Dan Gopstein
New York University
New York, USA

Anne-Laure Fayard
New York University
New York, USA

Sven Apel
Saarland University, Saarland Informatics Campus
Germany

Justin Cappos
New York University
New York, USA

ABSTRACT

Atoms of confusion are small patterns of code that have been empirically validated to be difficult to hand-evaluate by programmers. Previous research focused on defining and quantifying this phenomenon, but not on explaining or critiquing it. In this work, we address core omissions to the body of work on atoms of confusion, focusing on the ‘how’ and ‘why’ of programmer misunderstanding.

We performed a think-aloud study in which we observed programmers, both professionals and students, as they hand-evaluated confusing code. We performed a qualitative analysis of the data and found several surprising results, which explain previous results, outline avenues of further research, and suggest improvements of the research methodology.

A notable observation is that correct hand-evaluations do not imply understanding, and incorrect evaluations not misunderstanding. We believe this and other observations may be used to improve future studies and models of program comprehension. We argue that thinking of confusion as an atomic construct may pose challenges to formulating new candidates for atoms of confusion. Ultimately, we question whether hand-evaluation correctness is, itself, a sufficient instrument to study program comprehension.

CCS CONCEPTS

• **Software and its engineering** → *Software usability.*

KEYWORDS

Program Understanding; Think-Aloud Study; Atoms of Confusion

ACM Reference Format:
Dan Gopstein, Anne-Laure Fayard, Sven Apel, Justin Cappos. 2020. Thinking Aloud about Confusing Code: A Qualitative Investigation of Program Comprehension and Atoms of Confusion. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409714

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409714

1 INTRODUCTION

Previous work on atoms of confusion [11] introduced a methodology for discovering, measuring, and validating programmer misunderstanding in a precise way. An *atom of confusion* is the smallest snippet of code that will often confuse a programmer as to what the code's output is. Previous work measured correctness rates of programmers hand-evaluating confusing snippets and compared the rates to those for functionally equivalent code hypothesized to be less confusing. Between the minimality of the code snippet and its comparison against a control, the research on atoms of confusion was designed to be both precise and accurate. Gopstein et al. [11] applied this protocol in an experiment with 73 participants and analyzed the results with modern statistical techniques.

The study performed by Gopstein et al. was significant in that it was empirical, objective, and quantitative. Code was found to be confusing or readily understandable based on experimentation, not theory; the observations were based on performance, not opinion, and the extent of confusion was able to be precisely quantified. Thus, the experiment was designed to maximize internal validity [20]. By using minimal code snippets, Gopstein et al. could be sure that they were only measuring precise code constructs. By using functionally equivalent code samples as controls, they were able to demonstrate a direct relationship between the code and programmer confusion.

Despite Gopstein et al.'s precision and accuracy in design, it can only tell us the outcome of programmers' performance, but not how or why they behaved that way. How can we know that the causes of confusion are those put forth by the researchers? How can we know that misunderstandings amongst multiple programmers are homogeneous. How can we even know that hand-evaluation captures all types of misunderstanding?

In short, Gopstein et al.'s strong focus on internal validity and objectivist rigor does not tell the whole story. We study the same fundamental code snippets and hand-evaluation protocols as Gopstein et al., but augment the setting by having programmers think-aloud as they participate, followed by a semi-structured interview and discussion. This unique perspective on an existing methodological framework allows us to understand and scrutinize existing work. Our experience with conducting a qualitative study after a quantitative experiment leads us to believe the original experiment could likely have been improved if a lightweight qualitative study had been performed as a pilot during the design of the original quantitative experiment.

Our study offers insights into previous results as well as several surprising observations that contradict previous assumptions, including:

- The origins of incorrect beliefs about semantics differ across programmers.
- Errors evaluating atom-containing code are often caused by other, unrelated aspects of the code snippet.
- Correct evaluation of a snippet does not mean a programmer understood its semantics.
- Our study reveals new types of potential atoms.

In Section 4, we outline descriptions of how and why programmers made mistakes or avoided doing so in surprising ways. This provides insight into how to more accurately interpret the results of Gopstein et al. as well as other hand-evaluation program comprehension experiments. In Section 5, we turn an eye to future research and propose potential improvements or new research questions.

A complete replication package for this study is provided at https://atomsofconfusion.com/2020-think-aloud. The goal of the replication package is to facilitate the understanding of our methods and observations as well as to encouraging others to perform similar studies of their own. While we provide the outcomes of our analysis, we recommend that anyone using this package stays open to new themes that might emerge. The package contains:

- Preparatory material, including all code snippets used and the scripts enlisted to assign them to subjects.
- Interview instructions, including a pre-flight checklist, meta-protocol and universal answer key used to increase the reproducibility of the semi-structured interviews.
- Raw Data, including anonymized transcripts and scans of the subjects' written notes from each interview.
- Analysis, including the labels assigned to the transcripts during open coding, and the codebook used in that process.

2 RELATED WORK

Atoms of Confusion. The core topic that inspired our study is the concept of the *atom of confusion*, a small snippet of code, empirically validated against humans and proven to be more confusing than other functionally equivalent code. Introduced by Gopstein et al. [11], the concept of atom of confusions relies on the mismatch between how programmers think the C programming language works versus how the specification defines it. Gopstein et al. define 'confusion' as when a programmer is asked to hand-evaluate a deterministic and syntactically/semantically valid piece of code, but reports standard output that is different from what is mandated by the language specification. Gopstein et al. argued that the most precise way to measure this phenomenon is with the smallest possible piece of code that can cause misunderstanding in a programmer while a similarly sized, functionally equivalent simplified 'pair' version of the code is able to be evaluated without error.

In the original study of Gopstein et al., 126 code snippets (each averaging 4 source lines of code), from 63 pairs, representing 19 proposed atoms were tested in a human subjects experiment with 73 participants. Each subject was shown 84 of the code snippets and asked to record what they thought was the output of each snippet program. Gopstein et al. were able to measure the correct/incorrect response rates for each obfuscated/transformed snippet pair and to determine which code patterns were truly more confusing than their counterpart. Of the 19 proposed atoms, 15 met the statistical significance required to be considered a confirmed atom.

Following the original studies, the notion of atoms of confusion has been shown to be common in practice and correlated with negative code quality indicators such as bug density and security vulnerabilities [12]. The concept has also been investigated with the open-source community through opinion surveys and pull-requests [17]. This line of investigation confirms that atoms of confusion are indeed confusing and prevalent across several dimensions. However, there has yet to be an investigation into the mechanism with which that misunderstanding occurs. Our research sets out to explain the phenomena observed in previous studies.

In an effort to expand the concept of atoms of confusion beyond just the C language, Castor adapted it to the Swift Programming language [4]. Castor used new methods of finding confusing patterns, such as measuring the infrequency of occurrence in large code bases and expert opinion. We propose that observing programmers in a think-aloud study is an acutely effective means of identifying specific sources of misunderstandings.

Qualitative Research in Software Engineering. With goals of understanding how confusion arises in programmers and improving methodologies for future research, we chose a qualitative method to explain previous results and explore potential new research designs. Despite the positives of Grounded Theory, we decided it was not a good fit for our study, as we were already familiar with pre-existing literature in the field, and the semi-structured nature of our inquiry was slightly too rigid to fully benefit from Grounded Theory. Still, we took many lessons both from primary sources of the technique [6, 9], as well as descriptions designed specifically for the software engineering field [22]. We used techniques recommended from these texts, such as continuous data analysis, (open) coding, and memoing. Perhaps the best high-level description of our style of study is described by Creswell as 'explanatory sequential mixed methods', where a researcher takes an existing quantitative study, "analyzes the results and then builds on the results to explain them in more detail with qualitative research" [7].

We modeled our research on qualitative studies in the software engineering field, combining elements as necessary to fit our needs. Röhm et al. used a hybrid design consisting equally of observations and interviews, with the "observation mainly targeting what developers do and the interview mainly targeting the motivation behind developer's actions" [18]. This blend of techniques was a natural fit for us, as we could reanalyze existing objective data, and then explain them based on the subjects' (subjective) reports. For the observation component, we chose a think-aloud protocol due to its ability to provide insights into active behaviors as they are being performed [5, 21]. This method is well suited to analyzing program comprehension tasks [3, 8]. However, public performance and an artificial code-reading environment have been linked to adverse affects on cognitive load [2], which may affect our results though not invalidate them altogether.

One of the most popular methods of analyzing qualitative data is first using open coding to develop descriptive labels for data, then using a hierarchical coding scheme to develop categories from these codes [6, 23]. This general technique has been used successfully many times in empirical software engineering and, more specifically, for studying program comprehension. Seaman gives a thorough overview of many of these methods [19]. In particular, Yamashita and Moonen used interviews to develop a taxonomy of difficulties during code maintenance [25]. We have employed a similar technique to develop our understanding of the ways programmers misinterpret atoms.

3 METHODS

We used a qualitative approach to help explain the results of Gopstein et al. Towards that goal, we borrow many elements of experimental design from their work, but augment them to facilitate a deeper, more flexible interpretation.

3.1 Source Code Selection

Gopstein et al.'s code snippets were divided by two dimensions: whether the snippet contained a (potential) atom of confusion or not, and which type of atom it contained or not. After their experiments, they divided the atoms into the categories of being validated as confusing or not. They tested 126 code snippets in total. For our work, 126 were too many to test, given the enormous effort involved in a think-aloud study. Instead, we selected 26 representative examples from the original set. We made sure in our corpus to include one of each type of atom, several samples without atoms, and complete obfuscated/transformed pairs of each of the proposed atoms that failed to meet statistical significance.

We wanted to investigate snippets that had shown confusing properties in previous experiments, as well as snippets that had been hypothesized to be confusing, but did not empirically demonstrate confusing properties in the original study. This meant that, in addition to verified atoms of confusion, we wanted to study hypothesized atoms of confusion that had previously failed to meet statistical significance. Additionally, we wanted to observe subjects hand-evaluating non-confusing snippets, as a baseline for comprehension of simple programs. Since many snippets with atoms removed are similar to each other, being small and containing a limited set of language constructs, we did not make an effort to study each atom snippet's direct transformation. For snippets that were hypothesized, but not confirmed as confusing, however, we did test direct transformations, since an overly confusing transformation has the power to confuse a subject and serve as a defective control.

Consequently, each subject was shown:

- 5 confirmed confusing snippets (C)
- 1 confirmed not-confusing atom transformation (NC)
- 1 hypothesized but not validated confusing snippet (HC)
- 1 transformation of the same HC snippet (HNC)

Snippets were assigned to specific participants in using these constraints, but otherwise distributed randomly by a script. The script that we used to determine snippet selection is included in our replication package.

3.2 Data Acquisition

The principle mechanism of our study is a think-aloud protocol with a combined semi-structured interview and discussion. We showed each subject 8 small code snippets of various degrees of confusingness from prior experiments. For each code snippet, we asked the subjects to hand-evaluate the code and report the standard output (results of the `printf` statement). We then asked the subjects to notate their confidence about their answer on a scale from 1 ('unsure') to 6 ('positive'). We also requested from each subject "While evaluating each program please speak aloud your reasoning, thoughts, and actions". The study leader remained in the room with the subject and audio recorded the entirety of the study. After each individual evaluation, the study leader first asked subjects why they chose the level confidence they selected, and then proceeded to ask clarifying questions about any ambiguous comments made by the subject. The study leader used a meta-protocol to guide their interactions. The meta-protocol recommended specific situations to look for, questions to ask in these scenarios, and canned answers for common questions. At this point, the study leader attempted not to provide any information to the subject. After all 8 code snippets were finished being evaluated, the subjects were given a questionnaire that asked about subject demographics, previous experience, programming language preference, and perceived proficiency, among other things. After the subject completed the questionnaire, the study leader stepped the subject back through each of the completed code snippets, this time asking more probing questions designed to start a back-and-forth dialog. At this point, discussions were collaborative and iterative; perhaps the study leader would ask a clarifying question towards the subject, or the subject would ask for an explanation from the study leader, and the discussion would proceed from there to identify the root causes and processes that led to uncertainty or misunderstanding. Any error in evaluation or understanding that the study leader identified was dissected both by the study leader and the subject until the subject had a complete understanding of the mistake they had made, and they had communicated their understanding of their original thought process and their misconception.

3.3 Subjects

We selected subjects to represent programmers along a broad range of experience, recruited from three general groups. Ultimately, we interviewed 5 students, 4 professional C++ application developers, and 5 professional authors of a popular C++ library. All subjects identified as male, though this was not an intentional aspect of the study design. A list of our subjects can be found in Table 1.

All student subjects were recruited from the computer science masters program of NYU. All other subjects were recruited from a single large North American Web technology company. Unsurprisingly, there is a much stronger preference for C++ among the subjects who work professionally with the language. Professionals also tend to be older, have more experience, and have higher self-perceived proficiency with programming.

3.4 Analysis

The analysis began as the interviews themselves, as the study leader kept continuous notes while each subject spoke. After each interview, the study leader would go back to summarize and reframe each set of observations. This included writing memos on developing patterns and theories as evidence for them grew. All-the-while these memos continued to be compared and juxtaposed against the newer data that continued be collected.

Table 1: Programmers recruited to participate in the study

Subj.	Group	Age	Preferred Language	C/C++ Proficiency	Years Progr.
4168	Student	22	C++ / Python	4	15
3316	Student	22	Go	3	4
4281	Student	24	C++	4	7
9112	Student	21	JS / Kotlin	3	6
3787	Student	22	Python	4	7
6061	C++ User	32	C++	5	27
8888	C++ User	35	C++	5	29
4304	C++ User	53	C++	5	39
1879	C++ User	48	C++	5	34
1157	C++ Librarian	37	C#	6	20
7640	C++ Librarian	29	C++	6	16
4642	C++ Librarian	53	C++	6	33
8697	C++ Librarian	22	JS / C++	4	5
1867	C++ Librarian	29	C++	5	9

Subsequently, subjects' handwritten notes were digitally copied and saved for analysis. We generated transcripts from the audio recordings of the interviews. We read these transcriptions, alongside the subjects' written notes, using open coding to label patterns that appeared in subject responses. Open coding, as described by Corbin and Strauss [6], is "Breaking data apart and delineating concepts to stand for blocks of raw data. At the same time, one is qualifying those concepts in terms of their properties and dimensions". This involves identifying interesting/repeated/surprising facets of a text and marking them for later analysis. From these codes, we formed higher level groupings of similar concepts. From these, patterns were induced that form the basis of our observations, hypotheses, and suggestions that follow in this paper.

The analysis was performed in several batches, intertwined with data collection as new subjects were recruited and interviewed. As one of the goals of this study was to explain Gopstein et al's results, we began by recruiting, interviewing, and analyzing student subjects, to obtain a sample similar and comparable to the original sample. We used a three-fold analysis strategy that allowed us to remain sensitive to insights that emerged from data [9], while focusing on the questions that emerged from Gopstein et al's original study. Once the first set of interviews was completed, the study leader read the transcripts to define emerging recurring themes across subjects. After discussing with the other authors this first list of common patterns, the study leader read the interviews a second time to inductively generate codes through open coding of our data. Whenever a new code was developed, it was added to a running codebook, which lists every type of code used, along with a brief description of when it is applicable. The full codebook as well as each code's application is included in our replication package online.

Since codes were generated dynamically during open coding, the coding process was performed iteratively. Any codes that were generated after the coding of a transcript were then retroactively applied to all transcripts that were previously coded without it. This process was done in two phases. Whenever the last subject of a group was coded, all the previous transcripts from that same group would be re-evaluated to add the newly-created codes. For example, when the last student transcript was coded, all previous students' transcripts were re-analyzed to see whether the new codes were applicable. Finally, when all 14 transcripts across all subject groups (student, C++ application developer, C++ librarian) were analyzed, they were all read and recoded once more, this time with the complete set of codes.

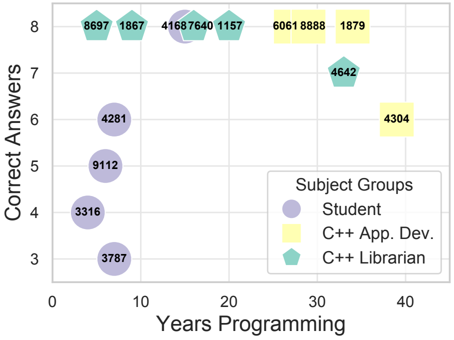

Figure 1: Subjects' performance relative to C/C++ experience

We then engaged in a third phase of analysis in which the study leader compared and contrasted across codes, and collapsed and combined them into higher-level categories. The study leader had regular meetings with the other authors to discuss and agree on the definition of these high-order concepts. In the rest of this paper, these concepts are presented with specific quotes and interactions.

3.5 Descriptive Statistics

While not the focus of this work, we provide a quantitative overview of the collected data because it gives important context for what follows. In total, there were 8h18m of recorded audio, consisting of about 82 000 spoken words broken into roughly 3040 blocks of speech by either the subjects or the study leader. There were 144 types of labels applied to 1740 phrases from the interviews. The most common themes in the text were 'Unsure', 'Correct Semantics', 'Simple'. Examples of less frequently occurring labels are: 'Maintainability', 'Previously Used', and 'Unfamiliar Syntax'.

There were 17 code snippets evaluated incorrectly. 14 errors were committed by 4 students, 2 by a single C++ application developer, and 1 by a C++ librarian, indicating that the students made more errors than professionals, but the different types of professionals were not well distinguished. A full accounting of errors-per-subject is depicted in Fig. 1. The types of snippets that generated errors largely followed what was previously reported by Gopstein et al., albeit with less fidelity. There were no 'transformed' (non-atom) snippets that were evaluated incorrectly, with all errors coming from snippets that were designed as obfuscated.

4 MECHANISMS OF MISUNDERSTANDING

In this section, we analyze and discuss our observations pertaining to the causes of programmer misunderstandings while evaluating atoms of confusion. We explore high-level patterns induced from individual examples discussed with our subjects, specifically:

- A taxonomy of different types of errors
- Causes of errors beyond atom of confusions
- Cases when subjects evaluated code correctly despite being mistaken about semantics

The discussions introduce a generalized pattern inferred from the data, and also give specific examples in the form of quotes taken from the subjects. Whenever quotes are given, they reference the subject and the associate code snippet in the form of [subject-id:snippet-id]. For example, if subject 1234 were describing snippet 56, we would write "quote about code" [1234:56].

4.1 Taxonomy of Confusion

Gopstein et al. were careful to delineate types of confusing code that were and were not relevant to their investigation. Specifically, they excluded code that contained non-determinism, non-standard features, high computational load, or reliance on a complex API. The rationale was that they were interested in studying situations in which a programmer would make an error based only on a flawed understanding of a language's semantics. To test this reasoning, we categorized every incorrect hand-evaluation and attributed a primary (and sometimes secondary) high-level cause. These causes naturally fell into four general groups of errors: *unfamiliarity*, *misunderstanding*, *language transfer*, and *attention*. The first three would likely be considered 'in-scope' for Gopstein et al., while attention-related errors would likely fall into the category of excluded types of confusion. Table 2 lists each incorrectly evaluated snippet, along with which high-level category of error it was assigned to. Next, we describe the categories in detail.

Unfamiliarity. The subject appears to have never encountered this phenomenon before, or at least does not recognize it in its current form. Consider the following example:

```
int V1 = 013;
```

> "I assume the int is going to be thirteen, because it doesn't have to represent that zero in memory" [3316:105]

In this case, the subject was not aware that a leading zero means the literal represents an octal value, and is stored with a value equivalent to the decimal number 11, in this case. There is no evidence that the subject had ever seen this notation before and was then left inferring the semantics of this notation based only on his own reasoning. Originally, the subject considered that a leading zero might cause an error, but ultimately he suggested that ignoring the leading zero might make sense for purposes of being able to parse fixed-width integers from data files.

All these behaviors are characteristic of unfamiliarity:

- No evidence of previous association of the construct
- Often questioning the code's validity
- Using logic to construct potential semantics

Misunderstanding. The subject is aware of the construct, but uknowlingly attributes incorrect semantics to it, as in this example:

```
if (++V1 || ++V2) ...
```

> "This is pre-increment and uh, what I remember from what I studied is that, you know, before the statement is executed, this will be incremented" [4281:79]

In this example, the subject recognized the pre-increment operator and noted that the underlying variable would be incremented. The subject believed that first incrementation would happen, and then the logical expression would be evaluated. Actually, it works the other way, the disjunction operator (`||`) must first evaluate its left operand, and only if that is true, it will evaluate the right operand, incrementing its variable's value. Under the subject's conception, `V2` would always be incremented, which is not accurate.

Unlike unfamiliarity, when a subject misunderstands:

- They have seen the construct before
- They purport to have an understanding of it
- Their understanding is somehow flawed

Language Transfer. The subject has seen similar code before, but only in another programming language. From the other language, they guess what the semantics might be in C:

```
int V1 = 013;
```

> "We have like different number base, but for int. I know Python has something like it, but in C, I'm not sure. [In Python, the 0 in front] means nothing." [9112:105]

In this example, the subject recognized the form of the octal literal, however, he was unsure whether or not it applies to C. He described how similar code works in Python, and then ultimately decided the `0` was likely to be ignored since that is what he believed happens in Python. (N.b. in Python 2, `013` is interpreted a decimal 11; in Python 3, `013` is a parse error).

Language transfer is easily identified when the subject:

- Mentions their understanding is derived from their knowledge of another programming language

Attention. The subject verbalizes correct semantics, but appears to forget or ignore an important piece of the computation:

```
printf("%d %dn", V1, V2)
```

> "V2 is going to equal 1 and V1 is going to equal 2 after that expression. And so it's going to print out one, two" [4304:61]

In this example the subject accurately described the state of the variables (`V1` is 2, and `V2` is 1), but then immediately went on to say that the variables were printed with their values reversed. There was no explanation for this what-so-ever, and the attribution of 'attention' is somewhat of a guess on the behalf of the study leader.

A misunderstanding is considered an attention problem when:

- The subject appears to understand the semantics of the involved constructs
- There is no other obvious cause of the problem

Table 2: Incorrectly evaluated snippets. Including error description, categorization, and unexpected additional atoms that also appear

Subject	Snippet	Atom	Mistake	Category	Other Atoms
3316	45	*Pointer Arithmetic*	Array as header pointer / pointer vs. value	*Unfamiliarity / Unfamiliarity*	
3316	19	*Pre-Increment*	Failed to increment variable	*Misunderstanding / Attention*	
3316	11	*Operator Precedence*	Boolean value of integer	*Language Transfer*	*Implicit Predicate*
3316	105	*Literal Encoding*	Leading zero is ignored	*Unfamiliarity*	
4281	79	*Logic as Control*	`++x` operates before statement	*Misunderstanding*	*Pre-Increment*
4281	71	*Preprocessor in Stmt*	`if` determines macro definition	*Unfamiliarity*	
9112	61	*Comma Operator*	Mistook = for == / Comma operator	*Attention / Unfamiliarity*	*Assignment as Value*
9112	105	*Literal Encoding*	Leading zero is ignored	*Language Transfer*	
9112	79	*Logic as Control*	Forgot short-circuit / thought it didn't apply	*Attention / Misunderstanding*	
3787	11	*Operator Precedence*	Boolean value of integer	*Unfamiliarity*	*Implicit Predicate*
3787	71	*Preprocessor in Stmt*	If selects preprocessor	*Unfamiliarity*	
3787	109	*Omitted Curly Brace*	Boolean value of integer	*Unfamiliarity*	*Implicit Predicate*
3787	37	*Macro Precedence*	Pre-evaluated macro contents	*Unfamiliarity*	
3787	85	*Repurposed Variable*	Boolean value of variable (not integer)	*Unfamiliarity*	*Implicit Predicate*
4304	61	*Comma Operator*	Switched argument order	*Attention*	
4304	79	*Logic as Control*	False operand causes short-circuit	*Misunderstanding*	
4642	115	*Type Conversion*	Float to integer is a rounding conversion	*Misunderstanding*	

Summary. We have defined four categories of error that describe the mistakes that we saw in our study. We do not expect they are exhaustive of all comprehension errors. However, they capture the essence of the issues that we happened to witness.

4.2 Unexpected Cause of Error

The very premise of earlier work on atoms of confusion work relied on the implicit assumption that different programmers misinterpret various examples of confusing code in the same, predictable way. For example, an atom of confusion might be called *Operator Precedence*, and any time a subject makes an error evaluating this piece of code, the specific cause of confusion is assumed to be directly due to misinterpreting precedence of the infix operators in the snippet. Despite the tight controls implemented by Gopstein et al., there was no way to validate that assumption. In our work, we see evidence both in favor and against this assumption. One example that shows an incorrectly attributed source of confusion happens with a code snippet of the *Operator Precedence* atom mentioned previously:

```
int V1 = 0;
if (0 && 1 || 2) {
  V1 = 6;
} else {
  V1 = 3;
}
printf("%d\n", V1);
```

The part that was assumed to be confusing was the precedence of the logical operators in expression `0 && 1 || 2`. Two (both students) of our 14 subjects incorrectly evaluated the logical expression above, however, the root cause of both errors were unrelated to operator precedence. Subject 3316 said "you cannot evaluate two to true" [3316:11], implying he did not know that non-zero values are evaluated as truthy by a logical operator in C. Subject 3787's confusion went slightly deeper: "so will the variable will be affected by this or it will just completely skip this" [3787:11], implying that he believed all logical conditions must reference a variable. While this code snippet was obviously confusing to these two subjects, it happened not to be the reason that was identified by previous research.

To determine which of the errors made in our study were due to the reasons assumed and which were due to unrelated misunderstanding, we analyzed each incorrectly evaluated snippet and attributed a simplified cause as to why each subject erred while hand-evaluating a snippet in Table 2.

Of the 17 incorrectly evaluated snippets, 10 appeared to be caused solely by the stated atom of confusion, as described in Gopstein et al. In 7 cases, the cause of the confusion was either partially or solely due to some other factor, potentially another atom or atom candidate. Subject 4304's error in snippet 61 appeared to be caused only by a lapse of attention, and is likely not caused by any atom of confusion at all. Several of the interactions observed during the incorrect evaluation of other snippets provided evidence for multiple new types of atoms not previously identified. This is discussed in more detail in Section 5.1 (*Potential New Atoms*).

Summary. Despite causing confusion, atoms of confusion do not necessarily cause confusion for the reason implied by the atom's title. This may be due to insufficient 'minimality', and perhaps the snippets could be written more carefully to avoid this pitfall.

4.3 Correct for Wrong Reasons

There is a subtle problem with only collecting the output of the subject's hand-evaluation. It is only possible to measure whether or not their evaluation resulted in the correct output, not whether they evaluated the code following the correct semantics. In this work, we observed several scenarios in which subjects reported correct output, but did so in a circuitous way. In Table 3, we collected every example of a construct that was evaluated correctly, but for the wrong reasons. The table lists which piece of code was evaluated with wrong semantics, what the supposed or explicit misunderstanding was, and whether or not the subject proceeded to ultimately evaluate the rest of the snippet correctly.

In our example on *Operator Precedence*, there were no wrong answers that exhibited the expected misunderstanding of precedence rules. This could be because precedence rules for logical operators are not confusing. However, there is also another explanation provided by our investigation. Several subjects did express a lack of understanding of operator precedence, however, due to the ordering of the expressions involved, they would accidentally correctly evaluate the snippet anyway. Take, for example, subject 8697 who explained "If I recall correctly, the operator precedence is such that And and Or are the same. I don't believe either one of them takes precedence over the other, and so it's just left to right" [8697:11]. This understanding of the language semantics is generally incorrect, however, the specific example enabled the subject to still evaluate the example correctly. Had the snippet prompt instead asked the subject to evaluate `2 || 1 && 0`, it is likely that he would have evaluated it incorrectly assuming the parenthesization would be `(2 || 1) && 0` instead of the actual parenthesization: `2 || (1 && 0)`.

Unlike a simple error that leads to an incorrect output, a correct response that arose from an incorrect evaluation often necessitates multiple misunderstandings: one to initially generate an incorrect response, and another to steer the evaluation back to being correct. For example, in snippet 49, subject 4281 did not understand the parsing rules of the statement `int V2 = V1 == 3 ? 2 : 4;`, however, since he believed that == was only valid to be used inside a predicate clause (for example, the first operand of the conditional operator), the two misunderstandings canceled out, and left the subject evaluating the code correctly.

Summary. The implication for these types of errors on previous work is significant. Misunderstandings like these may not result in an incorrect response in a correct/incorrect output experiment like the one of Gopstein et al.. So, if an experiment intends to be measuring misunderstanding, it may well be underestimating the amount.

5 IMPLICATIONS AND PERSPECTIVES

During the course of this study, it became clear that many assumptions that were taken for granted in previous work may not always hold true. In this section, we discuss potential new areas to investigate or improvements on old designs for new studies going forward. We address four specific facets of the previous work from a new perspective:

- New candidates for atoms discovered through observation
- Limitations of the model of atoms of confusion
- Potential pitfalls in hand-evaluation experiments
- The role of qualitative methods in quantitative work

5.1 Potential New Atoms

Gopstein et al. proposed 19 candidate code patterns as potentially confusing relative to functionally equivalent code. Of these, 4 failed to meet statistical significance; 15 patterns were validated as atoms of confusion. Despite this existing corpus of known confusing code patterns, it is likely that there still exist more minimal patterns of code that are confusing to programmers. We list here several constructs that appear to be confusing based on our observations; they may prove to be atoms of confusion upon further validation.

Uninitialized Variable. Several of the programs included in our study declare variables and do not initialize them immediately, only assigning values to them through the course of the program. Each of these variables, being local in scope, begin life with an indeterminate value and only have a known value once it has been assigned. In each snippet, the value of every variable is assigned before it is read. Therefore, no code has unspecified or undefined behavior. However, there were dozens of mentions of this concern from subjects: "The others are actually in an indeterminate state at the moment because we have no guarantee of like in Java they'd be zero initialized, but this is, C you go to hell and you die" [1157:85]. In most cases, subjects understood that there was no unspecified behavior, but that it was still bad practice: "The intention is to, to not have the, the like declared, declaration of an int that's not defined because it's just a, uh, it's a vector for bugs" [1867:50]. We noticed, however, several misconceptions around the semantics of uninitialized values that either caused mistakes or likely would have in a different context. For example, subject 9112 incorrectly evaluated snippet 61 in part due to this misunderstanding. He recalled his understanding "From my experiences [the default value will be] either minus one or to the smallest number from the system" [9112:61] and then proceeded to use these values in the snippet. Other subjects still managed to navigate the code correctly despite equally unfounded beliefs: "It depends on the flavor of C that you're actually working in. Um, actually I don't know for even today, I'm not sure if like the default, initialization... I think that since C++11, they made sure that these are default initialized, so they'd be zero. But that's not guaranteed in the early versions of C" [4304:61]. In this case, C++11 does not allow for automatically initialized integers in a non-static/non-global context [14], and that has been the case in the standard, at least, since ANSI C [15].

Given these misunderstandings, it is likely that code containing uninitialized variables is confusing to programmers. The original work on atoms of confusion specifically avoids testing code with unspecified/undefined behavior, however, making the testing of this potential atom a little challenging. One interesting fact that can be tested is that integers in a static or global context are (and have been since the ANSI standard) initialized to `0`. So, one potential snippet of atom code could be:

Atom Candidate: Uninitialized Value

```
int x;
int main() {
  printf("%d", x);
}
```

Obfuscated

```
int x = 0;
int main() {
  printf("%d", x);
}
```

Simplified

In this example, the standard specifies that `x` is initialized to `0` and therefore output is `0`, however, several of our subjects would likely believe the behavior is unspecified or undefined.

Modulo Operator. In snippet 1, an example of the *Implicit Predicate* atom, the modulo operator (written as `%`, and often verbally abbreviated 'mod'), was used as a generic non-logical expression in the condition of an `if` statement like: `if (10 % 3)`. While the name and concept of the modulo operator was always known by our subjects, the precise semantics and evaluation were often questioned. One

Table 3: Every evaluation that was correct, but for the wrong reasons

Subj.	Snippet	Construct	Misconception	Ultimately
8697	1	Truthiness of `10 % 3`	Knew `10 % 3` was non-zero, but did not know its value	Correct
8697	11	Precedence of `1 && 2 || 0`	`&&` and `||` have the same precedence	Correct
3316	11	Precedence of `1 && 2 || 0`	All operators get evaluated from left to right	Incorrect
4281	79	Incrementation and short-circuiting of `++V1 || ++V2`	(a) Both operands get evaluated; (b) `++3 == 3`	Incorrect
4281	49	Precedence of `V2 = V1 == 3 ? 2 : 4`	"double equal-to can only be there for the condition check"	Correct
9112	61	Expression value of `(V2 = 1, 2)`	"[assignment] just like return like nothing. So only the remaining value is 2"	Incorrect

subject, in particular, knew very well what the operator was at a high level, but struggled to actually calculate its value. "I mean it's like, obviously it's modulo operator, so it's like kind of close to the remainder... [subject elaborates more, but then pauses for 31 seconds trying to calculate the value of the operation] oh, okay. Actually now that I think about it. I don't really care what the value is, I guess cause it's like it's just a conditional, so it's like you can convert it to Boolean. Um, so in this case, uh, it does not evenly divide into three is not evenly divided into 10. Uh, so there is a remainder and of non zero value. So it's true" [8697:1]. After being prompted by the study leader, the subject explained that, despite knowing the high-level description of the modulo operator, he struggled actually calculating its result. The study leader pushed him to try to calculate the value again, to which the subject resorted to counting on his fingers, and still feeling very unsure of his result: "one, two, three. One, two, three, four, five, six, seven, eight, nine, 10. So that would be... Oh interesting. Actually. Not sure what though, so it's starting at. Would that be one or zero? I mean it's, it can't be zero because it's doesn't evenly divide" [8697:1]. Despite evaluating every snippet correctly, had the study leader asked subject 8697 to evaluate `10 % 3` and report its value, he would not have been able to do so.

Even beyond the computational ability required of the modulo operator, several subjects also agreed the semantics of the operator was unknown to them when the operands were non-positive. "It's the mod operator. Could have gotten interesting if there were negative or events that I had to think about it... Like specifically a negative divisor would like I'd never think about that case... And I guess they design it so that like if you flip the sign at both of them, it should do the same thing I think was there, uh, I don't remember. There's a couple of logical ways to do it." [8888:1]

Given these comments, it is likely that the modulo operator is itself confusing. Both the example given, for computational reasons, but also usages with other operands. Below are examples of the modulo operator being used with operands of various signs, which may be confusing as well.

Atom Candidates: Modulo Operator

Case	Expression	Value
Two Positive Operands	`10 % 3`	`1`
Negative Left Operand	`-10 % 3`	`-1`
Negative Right Operand	`10 % -3`	`1`
Two Negative Operands	`-10 % -3`	`-1`
Zero Left Operand	`0 % 3`	`0`
Zero Right Operand	`10 % 0`	undefined

Octal notation (two reasons). Snippet 105 tests whether programmers understand how to print an octal literal integer as a decimal value. The relevant code is:

```
int V1 = 013;
printf("%d\n", V1);
```

There were two incorrect evaluations of this snippet, both by students. One of the subjects, 3316, was not at all familiar with the leading-zero syntax and assumed it would be ignored: "Um there's a leading zero there. So I assume the int is going to be 13, um, because it doesn't have to represent that zero in memory." [3316:105]. The other subject, 9112, knew that some languages represented octals in this way while others didn't and made the guess that C did not. Afterwards the study leader told 9112 that the leading zero did mean octal, and asked the subject to proceed from that point with the new knowledge. From there, the subject was still unable to get the correct result: "It's a base 8. And I think per this means that the output for the be a digit [sic], so it's 1 plus 8. It's 9." [9112:105].

Between these two subjects, we can see that there are two different types of misunderstandings, one regarding the meaning of the leading-zero notation, and the other about how to translate between octal and decimal numbers. Perhaps this indicates that there may be room to reduce the complexity of the code snippet further, by factoring it into two seperate more 'minimal' atom candidates instead.

Atom Candidates: Octal Notation and Conversion

Atom Candidate	Code	Value
Leading-Zero Notation	`printf("%o", 07+010)`	`17`
Decimal-to-Octal Conversion	`printf("%o", 15)`	`17`

String as Pointer. In C, a string is represented as a pointer to a series of sequential characters in memory, where the last character is indicated by a null byte. This is why strings are declared as `char *` or "pointer to a character". It would appear, though, that the specialized nickname 'string' or something about its conventional representation may obscure its underlying structure to some programmers. Snippet 91 was designed to test whether programmers understood that the arguments to the array-subscript operator (square brackets `[]`) can be swapped without affecting the program behavior. One of our subjects nearly evaluated the snippet incorrectly not due to misunderstanding the subscript operator, but instead because he did not realize strings were just pointers: "I have seen it on some of the class material once where, you know, that was, you know, professor was just going through it and he said that this or that wouldn't matter. And I, I did try it with an `int` because I just wanted to check to be sure, it did work, but you know, I kind of initialized an array before and then I tried it like this with the index and the, you know, base pointer, but I haven't done that with strings... but this is not a base pointer, but I do understand it's a string and I'm really shaky when it comes to strings in C" [4281:91]. It might be valuable to measure the confusingness of the pointer representation of strings independently, to learn how well subjects understand them:

Atom Candidate: String as Pointer

```
void main() {
  char *x = "abc";
  char y = *(x + 1);
  printf("%c", x);
}
```

Obfuscated

```
void main() {
  char[] x = {'a','b','c'};
  char y = x[1];
  printf("%c", x);
}
```

Simplified

Logical Operator Outside of Condition. The *Implicit Predicate* atom implies a perceived connection between conditional statements and logical operators. The atom shows that it is more confusing to use an `if` statement (for example) without an equality operator than with one. Conversely, if programmers have learned a conventional association between conditions and logical operators, perhaps the presence of logical operators in absence of an `if` statement or `while` loop is itself confusing.

Snippet 49, designed to test the *Conditional Operator* atom, also caused perceived ambiguity over the precedence of its operators in this statement: `int V2 = V1 == 3 ? 2 : 4;` Subject 4281, unsure of whether to parse the conditional expression as: `V1 == (3 ? 2 : 4)` or `(V1 == 3) ? 2 : 4` made a statement to this effect: "a double equal-to can only be there for the condition" [4281:49], and used this reasoning to help infer rules about operator precedence. Despite helping him correctly evaluate snippet 49, this understanding of the usage of the `==` operator is incorrect. Actually, the equality operator is used in an expression like any other, and can be used in any context that can receive an `int` (specifically 1 or 0). To subject 4281, however, the logical operator is inseparable from its conditional context, and would not have a value if it were: "It cannot be there for any other reason but for a condition... Nothing comes out of this" [4281:49].

It would appear that the conventional relationship between conditional statements and logical operands is so pervasive that some programmers have induced that it is by specification. Perhaps never having seen a logical operator outside of a select or looping statement, the programmer believes that it is not allowed to happen.

Atom Candidate: Logical Operator Outside of Condition

```
void main() {
  int x = 1 == 2;
  printf("%d", x);
}
```

Obfuscated

```
void main() {
  int x = 0
  if (1 == 2) {
    x = 1;
  }
  printf("%d", x);
}
```

Simplified

5.2 Challenges of the Model

Gopstein et al. introduced the notion of atoms of confusion as a way to simplify the process of program comprehension to a point where it can be precisely and reliably measured. The primary mechanism for gaining that precision was the minimality and atomicity of code snippets. Part of what our study is demonstrating is that talking about code and human comprehension in such dichotomous terms is not the most accurate model. We have seen several examples that undermined the concept of homogeneity of confusion, minimality of confusion, and evaluation as a precise proxy for comprehension.

Homogeneous misunderstanding. The way Gopstein et al. presented their 15 examples of confusing code patterns implied that misunderstanding was a trait inherent to the code itself, independent of the subjects on which they were validated. What we have seen in our study is that there is considerable variance across our subjects' types of misunderstandings during evaluation, even for the same code snippet. Sometimes this is affected by obvious factors such as experience, but sometimes it is subtle factors such as personal experience with a given construct or even the specific mnemonics a programmer has used to memorize a particular quirk of the language. Regardless, the more evidence we see, the less likely it seems that any particular code snippet is likely to be universally confusing or simple across all programmers. In fact, it is unclear whether or not specific constructs can be selected to be confusing for even a specific demographic, as each programmer is an individual with their own mastery and misconceptions unique to themselves.

One example is various programmers' misunderstandings of the Boolean value of integers when evaluated in a logical operation. In C, values are generally considered 'falsy' if they evaluate to `0` and 'truthy' otherwise. For example, logical conjunction is specified to behave as follows: "The `&&` operator shall yield `1` if both of its operands compare unequal to `0`; otherwise, it yields `0`. The result has type `int`" [13]. However, we observed several different beliefs regarding the Boolean value of integers.

Consider these three different examples of incorrect semantics:

- The only valid integers in a logical context are 1 and 0
  - "you cannot evaluate two to true, but you usually can evaluate zero and one... to other boolean types" [3316:11]
- Integers are not valid in a logical context, at all
  - "It's confusing because zero and one... what? What does, what's zero and one, what does? How would I... I don't know what it means." [3787:11]
- All integers are true because they exist
  - "I mean we're using zero here for the if, and zero is an integer. So it, it is true. I mean zero does exist." [3787:109]

Given that there is such a diverse set of misunderstandings around a single construct, it may not be appropriate to claim that there is a single source of confusion that causes all of them.

Mutually exclusive atoms of confusion, minimality, and atomicity. Part of Gopstein et al.'s restriction to 'minimal' code snippets meant that no example of an atom of confusion could contain any other confusing code beyond what was being tested. Unfortunately, this necessarily precludes the possibility that two potential atoms of confusion might share common conceptual ground. One example concerns expressions with side effects. Several responses in our study have indicated that code that mutates state as part of a larger expression is inherently confusing. Examples of this type of confusing code can be quite minimal, such as `a = b++` or `a = b()` (assuming `b()` produces side effects). The first example has been shown to be confusing as a *Post-increment* atom, while the second has not. Evidence from our study and other studies [1], however, suggests that the latter is a good candidate to be an atom. If, however, side effects are shown to be confusing and become a new atom in their own right, this makes existing atoms such as `a && b()` (*Logic as Control Flow*) or `a = (b(), c)` (*Comma Operator*) non-minimal, since they must contain the *Side-effecting Expression* atom. Given the confusing nature of short-circuiting, though, the *Logic as Control Flow* atom is likely to be significantly more confusing than the side-effecting expression it would contain. To put it differently, `a && b()` is confusing, it cannot have any part of it removed while retaining its level of confusion, and it cannot be written without a side-effecting expression. If side-effecting expressions are ever validated as an atom of confusion, this would suddenly cause `a && b()` to no longer be minimal, and therefore not viable as an atomic unit of confusion.

This contradiction might force a difficult choice between several valid atoms. Alternatively, it may point to a conceptual shortcoming in the definition of atoms of confusion. Perhaps the notion of atomicity is limited; the idea that there is a certain threshold level of confusion below which certain code is simple and above which it is confusing. If, instead, we accept that all code exists on a spectrum of understandability, and some code is easier to understand and some harder, there can no longer be a concept of a minimally confusing piece of code, because all but the smallest expressions can always be made smaller at the expense of some confusingness. Conversely, this lets us talk about several related, cooperatively non-atomic pieces of code that have varying levels of confusingness despite all sharing related constructs. For example, `a && b++`, `a = b++`, and `b++`. It is likely that each is confusing, but to differing degrees.

Evaluation vs. Comprehension. Perhaps the core assumption of research on atoms of confusion is the use of the correctness of source code evaluation as a proxy for program comprehension. In any specific context, this correspondence may remain valid, however, evidence from our study indicates that individual understanding is not monotonic. Section 4.3 (*Correct for Wrong Reason*) demonstrates that a programmer who can evaluate a small snippet of code may be able to understand the same code in another context. Even in circumstances where subjects correctly evaluated a snippet, occasionally certain assumptions they made possibly would not have been made in the context of a larger program, or would not have lead them to a correct answer in that case. Consequently, the relationship between evaluation results and claims of comprehension may not always be true for the types of code shown in these studies, let alone generalize to more typical code seen in larger code bases.

5.3 Threats to Validity in Previous Studies

Outside of confusing programming constructs that have potential to be atoms themselves, we discovered several unintentional features of the code snippet evaluation protocol. These raise threats to validity in code evaluation studies of the type used in Gopstein et al. Each of the examples below represent exceptions to the intended research questions of Gopstein et al. Some introduce confusion where they should not, others add mechanisms of inferring correct answers where they should not, and still others rely on non-standard features of the C language.

5.3.1 *`printf` is Confusing.*

The basis for hand-evaluation experiments is for the subject to report the standard output of the program in question. In C, the canonical method of generating standard output is with the `stdio` function `printf`. It is famous for its esoteric API that is rarely fully learned by users. Many subjects, especially the professionals, called this out as a potential source of confusion: "printf and the percent things I pretty sure I remember most of them and I'm pretty sure percent c prints the character, but I, I check the man page every time I have to use one those." [1879:91], and "we've got a float and we're printf'ing it with a percent d I never know about printf syntax, I never use it [nervous laughter]. "[1867:115]

According to the original description of atoms of confusion, certain types of confusion were excluded from the study, including API related confusion, which appears to be exactly what `printf` induces. Potential alternatives include providing documentation along with the experiment, or using C++'s `cout` object, which does not require a format specifier. Both of these solutions have downsides and neither is ideal, but a better solution is worth considering.

5.3.2 *Relying on the Correctness of the Example.*

The introductory text to our study said "Each program compiles and runs without error". A somewhat popular tool amongst subjects was to leverage that fact to make assumptions about the code. "I just remembered every program executes. So this, my thinking of this not working is wrong" [3316:37]. In practice, programmers likely will not have these types of assurances and cannot rely on the fact that arbitrary code is guaranteed not to produce an error.

5.3.3 *Variable Names.*

The snippets included in our study were designed not to include semantic beacons [24], which might telegraph the meaning of the code to the programmer before they fully comprehend how it works. Part of this process was the renaming of all variables to `V1`, `V2`, `V3`, etc. However, there were examples where perhaps the naming of the variables induced otherwise unlikely errors. For example, in snippet 61, variables `V1` and `V2` contained the values `2` and `1` respectively, effectively swapping the contents of the names and values. Subject 4304 made it clear that he knew which variable contained which value, however, when writing his response he accidentally swapped them. One obvious way this may have happened was by subconsciously writing the number contained in the name of the variable, rather than the value it contained. In future experiments, it may be beneficial to use a variable naming scheme that will not overlap with the stored information.

5.3.4 *Redefining Macros.*

The snippet included in our study to represent the *Preprocessor in Statement* atom, contains a macro redefinition. There are two `#define`'s of the same identifier in the same file, without an `#undef` between them. The C specification (Section 6.10.3.2) [13] specifically says this is invalid: "an object-like macro shall not be redefined by another `#define` preprocessing directive". And this was brought up by one of the subjects: "right here we have like, uh, a redefinition of both M1 and M2 to be different things. Um, and I'm not positive. I, I think that's not well defined. Um, and I think that should be a compiler error, but it could also be like overwrite to use the like the latter one" [7640:71]. There was some controversy here though as a second subject claimed it was valid "Like if you're defining a value in the preprocessor, and it's already set, but if you set it again, they will not warn you or whatever. It just lets you override it." [8697:71]. Interestingly, this behavior is documented as being allowed by GCC: "If a macro is redefined with a definition that is not effectively the same as the old one, the preprocessor issues a warning and changes the macro to use the new definition" [10]. It is unclear whether adding `#undef`'s would make the code less confusing, but the current version is not the best example from which to base conclusions about comprehension. Another factor worth considering is whether the C preprocessor, arguably an independent language, ought to be studied at the same time the C language proper, as the preprocessor is known to be used in ways that conflict with the underlying language [16].

5.3.5 Void Main. Presumably for brevity, all code snippets are written using `void` return types on the `main` function. C technically does not allow this (despite several compilers being permissive about it), and many professionals complained about it: "Right off the bat it's void main, uh, which is, I feel like I'm back in Java land. So, uh, it should be int" [8697:71], "The thing I was a little tweaked out by, is that void main" [8888:73], "Oh, so they are all void main. Oh, that's lovely." [1879:85], "Uh, main is void. That's a little bit strange to me" [7640:19]. Changing the type of the `main` function to `int` would also necessitate adding another line to each program, returning `0`. This return value might in turn complicate the subjects' understanding of what value to report as 'output'. Still, it is clear that `void main` is not an acceptable idiom in a study like this.

5.4 Improving Quantitative Experiments through Qualitative Pilots

The previous sections outline many flaws in an otherwise thoughtfully designed quantitative experiment. We contend that issues like these are not unique to this experiment, but likely manifest in different forms in many program comprehension experiments. Our experience illustrates how performing a qualitative study allowed us to make visible many issues that we did not originally foresee.

In our case, the quantitative and qualitative portions of these studies were conducted sequentially as two separate endeavors. The findings of our study may help design future quantitative experiments that explore some of the questions that emerged from this study. However, if Gopstein et al. had done a think-aloud study prior to designing their original experiment, they most probably would have found limitations similar to the ones we found and could have avoided them. Indeed, while Gopstein et al. report conducting a pilot study for their experiment, it was solely done to estimate statistical power to choose a sample size. Therefore, we recommend to researchers developing new types of program comprehension experiments to begin by validating their approach qualitatively prior to conducting their full quantitative experiments. While the effort involved in our study was significant, and probably prohibitive to researchers for whom it would be only a pilot test, we believe that it is possible to use similar approaches, yet in a less time consuming fashion. While we made sure to reach theoretical saturation when analyzing our data, as this was a stand-alone study, the main themes emerged after we analyzed the first half of the participants and could have informed the design of a quantitative study.

6 CONCLUSION

Atoms of confusion [11] are a model for conceptualizing, measuring, and comparing small, hard-to-evaluate patterns in source code. Previous research used experimental approaches to validate and quantify the effects of these patterns. To further contextualize the existing body of work, and to provide a richer description of subject's underlying reasoning, we performed a qualitative investigation of the same phenomenon. Specifically, we conducted a think-aloud study of programmers hand-evaluating atoms of confusion and their associated simplified code pairs. The observations made in this paper offer insights for future research on atoms of confusion as well as for more general hand evaluation studies on program comprehension.

As the second (qualitative) study in an explanatory sequential mixed-methods design [7], our study was designed to give context to the results presented by Gopstein et al. [11]. We wanted to go beyond the dichotomous data presented before, and describe more than just whether or not a programmer correctly evaluated a code snippet. We wanted to understand and describe how programmers evaluated code, what steps they took, what pitfalls they hit, and how they evaluated unknown constructs. We arrived at a common taxonomy that was able to naturally categorize many of the incorrect evaluations in our study and that shows that not all misunderstandings come from the same mechanism. We investigated responses given by subjects whose hand-evaluations were correct and learned that correct answers do not always imply correct reasoning on behalf of the subject. By using a qualitative research approach, we were able to see that, even for the data points that, in a quantitative study, would look absolutely correct, there was still significant confusion that would otherwise go unnoticed. Insights like these allow us to look back on previous research and understand it with new depth. We can see that hand-evaluation studies may be under-reporting the amount of misunderstanding, since correct responses are now shown not to necessarily imply complete understanding. Furthermore, we can see that, just because errors are being observed, they are not always due to the factors the experiment was designed to test.

Beyond understanding previous work, we identified several key factors that can help design similar experiments in a more rigorous way as well as entirely new studies to explore further ideas. Simple changes can help give more accurate results such as avoiding complex APIs like `printf`. We can expand the current understanding of what code is confusing by empirically validating more atoms of confusion. Finally, we have seen ways in which the current model of confusion is limiting. Defining confusion as a dichotomous event, something that either occurs or not, simplifies a very complex phenomenon. Instead, it may be valuable to consider a more flexible and complex model of confusion, one that is based on objective empirical evidence and quantitative measurement and thus accounts for arbitrary forms of code and the variation between programmers.

ACKNOWLEDGMENTS

We want to thank the participants in our study for their time We are also grateful to Gennadiy Civil for helping to organize and encourage this research. This work was supported in part by NSF grants 1444827 and 1513457 as well as DFG grant AP 206/14-1.

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, Blob and Spaghetti code, on program comprehension. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE, 181–190.

[2] Mahnaz Behroozi, Alison Lui, Ian Moore, Denae Ford, and Chris Parnin. 2018. Dazed: Measuring the cognitive load of solving technical interview problems at the whiteboard. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 93–96.

[3] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. 2011. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. ACM, 1–9.

[4] Fernando Castor. 2018. Identifying confusing code in Swift programs. In *Proceedings of the VI CBSoft Workshop on Visualization, Evolution, and Maintenance*. ACM.

[5] Elizabeth Charters. 2003. The use of think-aloud methods in qualitative research an introduction to think-aloud methods. *Brock Education: A Journal of Educational Research and Practice* 12, 2 (2003), 68–82.

[6] Juliet Corbin and Anselm Strauss. 2014. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications.

[7] John W. Creswell and J. David Creswell. 2017. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications.

[8] Beth Davey. 1983. Think aloud: Modeling the cognitive processes of reading comprehension. *Journal of Reading* 27, 1 (1983), 44–47.

[9] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter.

[10] GCC GNU Compiler Collection. (accessed March 4, 2020). *The C Preprocessor: Undefining and Redefining Macros*. https://gcc.gnu.org/onlinedocs/cpp/Undefining-and-Redefining-Macros.html

[11] Dan Gopstein, Jake Iannacone, Yu Yan, Lois Anne Delong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding misunderstandings in source code. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 129–139.

[12] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of confusing code in software projects: Atoms of Confusion in the wild. In *Proceedings of the International Conference on Mining Software Repositories*. ACM, 11 pages.

[13] ISO. 1999. *ISO/IEC 9899:1999: Programming Languages — C*. 538 pages.

[14] ISO. 2011. *ISO/IEC 14882:2011 Information technology — Programming languages — C++* (third ed.). http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[15] Brian W Kernighan and Dennis M Ritchie. 1998. The C Programming Language-ANSI C Version.

[16] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM, 191–202.

[17] Flávio Medeiros, Gabriel Lima, Guilherme Amaral, Sven Apel, Christian Kästner, Márcio Ribeiro, and Rohit Gheyi. 2019. An investigation of misunderstanding code patterns in C open-source software projects. *Empirical Software Engineering* 24, 4 (2019), 1693–1726.

[18] Tobias Röhm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *Proceedings of the International Conference on Software Engineering*. IEEE, 255–265.

[19] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572.

[20] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on internal and external validity in empirical software engineering. In *Proceedings of the International Conference on Software Engineering*, Vol. 1. IEEE, 9–19.

[21] Maarten W. Van Someren, Yvonne F. Barnard, and Jacobijn A. C. Sandberg. 1994. *The Think Aloud Method: A Practical Approach to Modelling Cognitive Processes*. Academic Press Inc.

[22] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the International Conference on Software Engineering*. IEEE, 120–131.

[23] Anselm L. Strauss. 1987. *Qualitative Analysis for Social Scientists*. Cambridge University Press.

[24] Susan Wiedenbeck. 1986. Beacons in computer program comprehension. *International Journal of Man-Machine Studies* 25, 6 (1986), 697–709.

[25] Aiko Yamashita and Leon Moonen. 2013. Towards a taxonomy of programming-related difficulties during maintenance. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 424–427.