On The Impact of Atoms of Confusion in JavaScript Code

Adriano Torres[a], Márcio Okimoto[a], Caio Oliveira[a], Diego Marcílio[b], Pedro Queiroga[c], Márcio Ribeiro[d], Edna Dias Canedo[a], Fernando Castor[e], Rodrigo Bonifácio[a], Eduardo Monteiro[f]

[a]*Computer Science Department, University of Brasília, Brazil*
[b]*Faculty of Informatics, Università della Svizzera Italiana, Switzerland*
[c]*Informatics Center, Federal University of Pernambuco, Brazil*
[d]*Institute of Computing, Federal University of Alagoas, Brazil*
[e]*Department of Information and Computing Sciences, Utrecht University, The Netherlands*
[f]*Statistics Department, University of Brasília, Brazil*

Abstract

Evolving legacy code is a challenging task, particularly when the code has been poorly written or uses confuse idioms and language constructs, which might increase maintenance efforts and impose a significant cognitive load on developers. For this reason, researchers have investigated possible sources of confusion in codebases, including the impact of small code patterns (hereafter atoms of confusion) that contribute to misunderstanding the source code written in statically typed languages such as C, C++, and Java. In this work, we investigate whether atoms of confusion identified in statically typed languages also confuse developers of a dynamically typed language. We use JavaScript as a representative example of dynamic programming languages and collect evidence from a mixed-method research effort: a survey, a set of interviews with practitioners, and an activity of mining open source JavaScript repositories (MSR). Our survey and interviews confirm that atoms of confusion lead to code that is hard to understand in JavaScript. Considering ten atom candidates, developers correctly predict the outcome of at least 15% more cases in code snippets where atom candidates are not present, compared to alternative versions that include the atom candidates. For five of these atoms, the difference is statistically significant (accounting for p-value correction). Effect size is large for four atoms and medium for one of them. In addition, our MSR effort reveals that atom candidates are frequent and used intensively in 72 popular open-source JavaScript systems. Four atom candidates appear in 90% of the analyzed projects, and two of them occur more than once for every 100 lines of code in the dataset. Altogether, our findings might help practitioners: (1) better understand the implications of atoms of confusion on understanding JavaScript code, (2) avoid writing code that is unnecessarily difficult to maintain, and (3) design program transformation tools that remove potential sources of misunderstanding in JavaScript code.

Keywords: code readability, program comprehension, program understanding, atoms of confusion, JavaScript code

1. Introduction

Developers are often confused while reading unfamiliar code. According to Hermans [1], that confusion can stem from three sources: (i) lack of knowledge, i.e., not knowing what an element in the program does; (ii) lack of information, i.e., not knowing how a program element works; and (iii) lack of processing power, i.e., the inability to combine all the elements in a program and make sense of its execution in one's head. Even small pieces of code can be confusing for developers [2, 3] at first glance.

For instance, consider the pair of JavaScript code snippets in Listing 1 (adapted from NervJS/taro). The code snippet on the left-hand side is arguably harder to understand than the one on the right-hand side. The former uses the logical AND operator (`&&`) beyond its lexical meaning [4] to determine the control flow of the program. Furthermore, the expression has a potential side effect, due to the use of the post-increment operator (`s++`). The snippet on the right-hand side uses an `if` statement for control flow and increments the value of variable `s` using an assignment, making it arguably easier to understand. Notwithstanding, confusion is a direct consequence of the experience of the person who is reading the code. A developer who customarily employs the idiom on the left-hand side may find it easier to understand.

```
1 res = properties.reduce((s, b) => {
2   b && s++
3   return s
4 }
```

```
1 res = properties.reduce((s, b) => {
2   if(b) {
3     s = s + 1
4   }
5   return s
6 }
```

Listing 1: Code snippet from NervJS/taro project (left-hand side) and an alternative implementation (right-hand size).

Recent work [5, 3, 6] has attempted to elicit and catalog simple code patterns and language constructs that often confuse developers when reading code and for which there are less confusing, functionally-equivalent alternatives. These confusing constructs and patterns are called "atoms of confusion" [3] when it is possible to experimentally ascertain that there is a less confusing alternative (otherwise they are called "atom candidates"). The snippet on the left-hand side of Figure 1 contains two atoms of confusion discussed in previous work [3] focusing on the C language: Logic as Control Flow and Post-increment. Previous work has identified atoms of confusion in two statically-typed languages, C and Java. According to the studies of Gopstein et al. [3], removing the atoms Logic as Control Flow and Post-increment from small C code snippets improved the ability of study participants to predict their outcomes by 41% and 34%, respectively. For small Java code snippets, Langhout and Aniche [6] report improvements of 53% and 46.27%, respectively. These results show that atoms of confusion may be **challenging** for developers using these languages. In addition, a study [7] with 14 large-scale open source projects written in C found that 4.38% of the lines of code have an atom

and their presence has a strong correlation with bug fixing commits and long code comments. These results show that atoms of confusion are **prevalent** in large systems. They are also **relevant**, even for experienced software developers.

The goal of this paper is to investigate whether atoms of confusion that were identified in statically typed languages also cause confusion to developers of a dynamically typed language. We leverage JavaScript as a representative example of dynamic programming languages. JavaScript's programming culture differs from that of languages such as C and Java, particularly due to its dynamic capabilities and weaker type system. At the same time, the language is not so different that previously identified atoms cannot be represented in it. We present the results of a mixed-method research effort based on a survey, on interviews, and on a mining software repositories study. First, the survey's instrumment asks the participants to predict the output of small code snippets, with half of them containing atom candidates. Subjects were organized according to a Latin Square design [8] to control for the experience of the subjects. To learn more about the misunderstandings in the snippets used in the survey, we also interviewed 15 experienced professional developers. Participants preferred the code snippets without the atoms in 70% of the cases. The frequency with which the atoms appear *in the wild* might reveal cases where developers should be more or less careful when writing JavaScript code. To measure the prevalence of the atoms, we mined popular JavaScript projects from GitHub and found that atoms frequently appear in practice. Our MSR effort reveals that atom candidates are frequent and used intensively in 72 popular open-source JavaScript systems.

Altogether, the main contributions of this paper are:

- A mixed-method research effort—based on a survey, interviews, and mining software repositories—about the impact of atoms of confusion on JavaScript (a dynamic language), controlling for subject experience.
- A comparison of the impact and prevalence of atoms of confusion in JavaScript with what has been previously reported for other languages (C, C++, and Java).
- A library and infrastructure to mine atoms of confusion using code queries (implemented in the .QL language). This library and infrastructure are publicly available and might help researchers and practitioners to automate program analysis at a large scale.

2. Related Work

The concept of program comprehension is central to software maintenance and feature development [9, 10]. The study of program comprehension in the presence of atoms of confusion was introduced by Gopstein et al. [7], who defined them as small code patterns that can verifiably lead to misunderstandings. In addition, for a code pattern to be considered an atom, there must be some alternative pattern or language construct that

is functionally equivalent and less likely to cause confusion. One example that occurs in many programming languages is the Change of Literal Encoding. For instance, the C code statement `printf("\%d", 013);` often leads programmers to predict the output to be `13`, even though the correct answer is `11`. This occurs because a leading `0` in a numerical literal indicates that the number is in base `8`, a fact that is not only unknown to less experienced programmers, but misleading even for seasoned developers. After formulating a list of 19 atom candidates, Gopstein et al. [3] were able to identify 15 code idioms and language constructs that present a statistically significant difference in comparative answer correctness when each of the 15 atoms was removed. The least confusing atom produced a 14% boost in prediction accuracy when it was removed, whereas the most confusing one showed a 60% accuracy increase. In a different work, Gopstein et al. [7] presented the results of a comprehensive study on the incidence of atoms of confusion in the wild, considering open-source projects written in C and C++ programming languages.

These previous studies [3, 7] have a great influence in our work. Even though JavaScript is a dynamic language that differs from C in a variety of aspects, it also has a number of constructs in common. In addition, the imperative aspects of the language are syntactically similar to C, e.g., assignments, conditional expressions, `if`-statements, pre and post increments and decrements, among others. On the one hand, this means that some of the atoms of confusion that exist in C programs may also occur in JavaScript code. On the other hand, differences between the languages and the surrounding programming cultures may lead to code patterns that are atoms of confusion in one language not being atoms in the other one. Compared to previous research, in this work we leverage the *Latin square* experimental design to control both participant experience and education better, while investigating the effect of the atom candidates in code comprehension. We present details about our setup in Section 3. Differently, previous research works (e.g., [3, 6]) use a *Randomized Partial Counterbalanced Design*, which does not *block* relevant characteristics of the participants.

Oliveira et al. [11] conducted an experiment using an eye-tracker with 30 participants involving three atoms in C. They found that code with atoms of confusion requires more time from participants to predict the output and more visual effort to comprehend. Gopstein et al., in another paper [12] further explored the original atoms by additionally conducting interviews and having programmers discuss among themselves the studied atoms. The authors argue in favor of complementing a quantitative analysis. Indeed, their study revealed findings such as a *"correct evaluation of an atom might not mean that a programmer understood its meaning"*. Here, we also complement the quantitative analysis with interviews.

Langhout and Aniche [6] derived a set of 14 atoms for Java and performed an experiment with 132 students. For seven atom candidates (out of 14), they report that participants are more likely to make mistakes in the confusing version of the code. Castor [4] presented a preliminary catalog of six atom candidates for the Swift programming language. Unlike JavaScript, in Swift, most of the atoms identified

by Gopstein et al. [3] are avoided by construction, e.g., it does not have assignments as values, increment operators, or macros.

Medeiros et al. [5] analyzed 50 open source projects written in the C language with the goal of evaluating 12 code patterns called "misunderstanding patterns" by the authors. Many of these misunderstanding patterns are either atoms of confusion or atom candidates [3]. This study shows that these patterns are prevalent; among these 50 projects, there are more than 109K occurrences of misunderstanding patterns. In order to gauge the relevance of these misunderstanding patterns, the authors sent 35 pull requests removing occurrences of these patterns to randomly-selected open source projects. The authors of the study received feedback for 21 of these pull requests and the maintainers of the projects accepted 8 of them (22.86%).

3. Methodology

The main goal of this research is to investigate the impact of atoms of confusion on JavaScript code comprehension. As such, in this paper we answer the following research questions:

(RQ1) What is the impact of atoms of confusion on the comprehension of JavaScript code?

(RQ2) Do JavaScript developers identify atoms of confusion as contributing to program misunderstanding?

(RQ3) What is the frequency of occurrence of atoms of confusion in practice (i.e., in open-source JavaScript projects)?

Answering these research questions has several implications. For instance, we can either generalize or refute the perceptions about atoms of confusion already discussed in the literature (goal of research questions (RQ1) and (RQ2)). In addition, answers to the third and fourth questions allow us to enrich existing catalogs about atoms of confusion and discuss how often they occur in practice. Answering these research questions also lays the foundations for the implementation of tools that can automatically transform code into cleaner versions. We conduct a mixed-methods study to answer these questions, including a survey, a set of interviews, and a study for mining JavaScript software repositories.

3.1. Survey Study Setting

In the first study, we explore the impact of atom candidates on understanding JavaScript code (goal of research question RQ1). To this end, we investigate whether or not code snippets that contain atom candidates (a) produce a higher misunderstanding rate when programmers try to predict their outcome and (b) require more time for programmers to predict their output. Since JavaScript and C have some constructs in common (Section 2), we first selected a set of 9+1 atom candidates: nine previously-validated atoms of confusion for the C language [3] that also exist in JavaScript programs plus one atom candidate that is

specific to the JavaScript language (Automatic Semicolon Insertion). Our supplementary material discusses and shows examples of the atoms we consider in our research.

Survey design

The design of our first study blocks two variables (developer experience and the code snippets) and considers two treatments: the presence or absence of atoms candidates within the code snippets. To achieve such a design goal, controlling the effect of experience and individual code snippets, we resorted to the *Latin Square Design* [8]. Using this design we create a 2 x 2 matrix in which each row represents a subject and each column indicates the set of code snippets. The design of each square (a replica) is such that no treatment is repeated in the same row or column. For example, considering that we have a set of 10 code samples $s_0, s_1, ..., s_{10}$, if a given subject (P1) is asked to predict the output of the code samples $s_0, s_1, ..., s_5$ that contain atom candidates, then, when answering questions about non-confusing code snippets, they will only be presented with non-confusing versions of the code samples $s_6, s_7, ..., s_{10}$. Furthermore, a given subject (P2), who constitutes the second row of our example square, will be asked questions about the non-confusing versions for $s_1, s_2, ..., s_5$, and will answer questions about confusing snippets for $s_6, s_7, ..., s_{10}$. By doing that, we guarantee that all versions of the code snippets are contained within each square, and that each configuration occurs only once within a square. Figure 1 offers a visual representation of the concept.

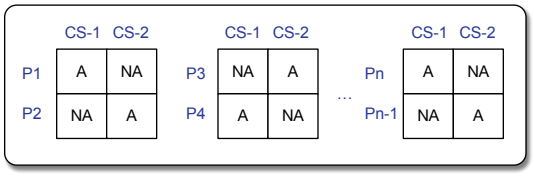

Figure 1: Latin square design. Each "square" corresponds to a replica in our study. Each replica comprises two participants (square rows, e.g., P1 and P2) and two sets of code snippets (CS-1 and CS-2). We randomly apply the treatments (atom or non-atom code) to the cells of the squares.

Having selected 10 atom candidates in our first study, we wrote 10 short programs, one containing each atom candidate. We call them the confusing versions of the programs. We also wrote corresponding short programs without the confusing idioms and constructs and we call them the clean versions of the programs. Overall, the survey employed 20 code snippets, 10 confusing and 10 clean. In order to reduce the cognitive effort, each subject was asked to predict the output of 10 code snippets, five confusing and five clean. The order in which the questions were presented was randomized. By doing this, we seek to minimize the chances of subjects being aware that the current listing they are analyzing contains (or not) atoms of confusion. That is, each respondent of the survey should indicate what would be the outcomes of the code snippets, some of them having atoms of confusion (while other code snippets do not). We measured answer correctness and the total time each participant needed to answer the survey's questions.

Survey Instrument

We implemented our survey as a web application. As part of this effort, we carried out an informal pilot whose main objectives were (i) to spot bugs in the application and in the data collection mechanism; (ii) to gain feedback from respondents about the user experience of the application; and (iii) to formulate an estimate about how long answering the survey would take on average. Undergraduate students and professional colleagues took the pilot survey. Some users reported layout defects, and many reported that the landing page did not explain the survey well enough. We also spotted minor issues with our routines to create and populate the Latin Squares.

We organized the survey in three sections. The first section aims to characterize the subjects, asking their age, education level, and programming experience. We also included a check button, whose checking meant users agreed that all collected data would be used solely for research purposes. In the second section, we presented to the participants a small set of instructions, where we explained how the survey worked and asked them to dedicate their attention to it. We stressed to participants the importance of not using any aids during the survey, such as online or console interpreters. For each question page, we kept track of whether or not the subjects switched windows.

The last section of the survey presented a sequence of ten questions, each containing a code snippet. For each question, there was a text box where the answer should be written. There was also an "I do not know" button, which, when clicked, led the subject to the next question. In our setting, "I do not know" was treated as a wrong answer. The code snippets were presented as images copied from a text editor, so as to demotivate respondents from resorting to external resources by copying and pasting the code into an interpreter. Upon submitting their answer for a particular question, the subject was automatically led to a similar page, containing another snippet.

We decided not to provide feedback about the time students took to answer each question. We also did not tell them whether their answers were correct or not. Our main concern was to avoid introducing bias for future respondents. Since we posted the survey on a social media platform, possible threats could have arisen if we gave respondents instant feedback.

Survey audience

We posted the survey on a JavaScript Reddit channel.[1] We explained our research purposes and asked developers of any level of expertise to take the survey. Within twelve hours, we collected more than 150 answers, populating more than 70 replicas of the Latin Squares. We collected significant data on time taken and discrepancies in answer correctness between confusing and non-confusing versions of the snippets. Some inconsistencies arose while building the squares, for instance, when a user quit in the middle of the survey.

[1] https://www.reddit.com/r/javascript/

We discarded from our study all squares that contained incomplete rows (a common approach for data imputation).

3.2. Interview Study Setting

To complement our initial survey, we performed semi-structured interviews with professional JavaScript developers, aiming to identify their perceptions regarding code snippets containing atoms of confusion (research question RQ2). We also asked each participant if they knew of any other JavaScript-specific construct or idiom that they regarded as being likely to make the code hard to understand (research question RQ3). In this section we detail the protocol we followed to conduct the interviews and to analyze the results.

Participant Selection

We invited the participants of the interviews using a snowballing technique. That is, starting from our network of contacts, we invited an initial set of candidates to take part in our survey. From this initial list, we asked for an indication of additional candidates. Our main selection criterion was that all participants should have been working with JavaScript in their daily professional activities. We invited a total of 17 developers, and 15 of them agreed to participate.

Interview Process

We conducted semi-structured interviews using web conferencing software. We recorded all the interviews with the consent of the participants. On average, the interviews lasted 26.29 minutes, with the shortest one lasting 14.59 minutes and the longest one 43.06 minutes. Two of the researchers conducted the interviews, and a third one listened to all the recordings to cross-validate the collected data. The interviews had three main parts. In the first one, we asked the developers the following demographic information: name, email, gender, level of education, current job position, JavaScript experience in years, and other programming languages they have worked with. Table 1 summarizes this demographic information.

In the second part of the interview our aim was to allow the subjects to describe their JavaScript experience, as well as to allow them to reveal any JavaScript constructs they regarded as innately confusing. This would allow us to identify potential atom candidates that are more specific to the JavaScript language. Examples of questions we explored in this section include: *Does JavaScript favor developers to produce code that is hard to understand?* and *Do you regard any particular construct or idiom of the language as especially confusing?*

In the third part of the interview, participants were shown the code snippets that were used in the first study. The snippets were presented in pairs, where each pair contained a clean and a confusing version. The participants were asked to evaluate which version of the code was easier to understand. To avoid introducing bias in the answers, the interviewers did not explain that one of the versions in each pair contained the atom

Table 1: Demographic information of the participants

ID	Education	JS Experience	Other Languages
P1	BSc Degree	9 years	Java, PHP, C, Go
P2	HS Degree	3 years	Python, Go, Dart, Lua, C++, C#
P3	BSc Degree	4 years	Java, C
P4	Undergraduate	3 years	Python, C, C++, Java, Go
P5	Master Student	3 years	Python, SQL
P6	Bsc Degree	15 years	Java, PHP, C, Python , Ruby, C#
P7	BSc Degree	6 years	C, C++, Java, Assembly, Kotlin
P8	PhD Degree	2 years	Java, Python
P9	BSc Degree	5 years	PHP
P10	Master Student	4 years	Java, C# and Python
P11	Master Student	4 years	Java, Erlang, C#, Cobol
P12	Master Degree	1 year	C
P13	Master Degree	13 years	Java, PHP
P14	Master Student	3 years	C, Python, Ruby
P15	BSc degree	2 years	Java, PHP

under investigation. Subjects were just presented the pairs of snippets and allowed to take the necessary time to decide on the most readable snippet.

Interview Analysis

We first transcribed each interview and then examined the broad distribution of the answers. Our goal was to build an initial understanding of the participants' perceptions with regards to the challenges to understand JavaScript code in general and JavaScript code with atoms of confusion in particular. We next followed with an open-coding procedure, highlighting the main themes and quoting the answers of the participants. We present these results in Section 5.

3.3. Mining JavaScript Software Repositories Setting

To understand how often the analyzed atom candidates appear in open source projects, and thus answer our fourth research question (*What is the frequency of occurrence of atoms of confusion in practice?*), we mined a set of GitHub open source repositories. To this end, we first collected the most popular GitHub repositories that are primarily written in JavaScript. We measured popularity using the project's stargazers. This metric, available through the GitHub API, represents the number of stars a project received from users of the platform. The same metric has been used in a number of previous studies as a proxy to estimate a project's popularity [13, 14]. We then selected the top 100 most popular repositories and removed projects that did not reach the first quartile of the distribution of lines of code.

After filtering out JavaScript project candidates, in the second step we built a curated dataset comprising the top 72 repositories. Examples of projects in this dataset include React, Node JS, and AngularJS.

Table 2 presents some statistics about the projects we consider in our research. The size of the projects range from small ones (5543 lines of code) to complex systems with more than 1 MLOC. All projects in our dataset have at least 1244 forks and at least 23 672 stars. We automated all the steps to filter, clone, and collect the statistics from the repositories using Python scripts.

We mined the occurrence of atom candidates from the repositories in our curated dataset using source code queries that we wrote using the CodeQL language [15]. CodeQL is an object-oriented variant of the Datalog language [16], and currently supports researchers and practitioners to query the source code of systems written in different languages (such as C++, Java, and JavaScript). We also automate the process of running the queries and exporting the results to a format that simplifies our analysis (and also the reproduction of this study). Finally, we computed some descriptive statistics to measure the prevalence of atoms of confusion in practice.

Table 2: Some descriptive statistics about the projects used in the MSR study

	Min.	Median	Mean	Max.
Lines of Code	5543	36 161.5	111 432.33	1 278 405
Num. of Forks	1244	6078	8906.24	68 849
Num. of Contributors	6	285.5	533.44	4047
Num. of Stars	23 672	34 990	46 919.51	310 935

4. Results of the Survey

Our first study investigates the impact of atoms of confusion while developers try to understand JavaScript code. We estimate this impact considering two perspectives: *misunderstanding rate* (number of wrong answers) and *time* (how long to provide a correct answer). We received full answers to our survey from 140 participants (a total of 70 replicas). All participants had taken at least some university course or hold a bachelor degree or equivalent, meaning the participants have had some level of formal education in programming. In addition, 21 participants hold a master's degree and three a doctorate degree. Considering the programming experience of our respondents, 19% have more than ten years of programming experience, 37% have between four and ten years of experience, 37% have between one and four years of experience, and 7% have less than one year of programming experience. Accordingly, we characterize the effect of atoms of confusion in JavaScript code taking into account the perceptions of both novice and experienced developers.

4.1. Misunderstanding Rate Analysis

Exploratory Data Analysis

As mentioned in Section 3.1, each participant in this study evaluated ten code snippets, from which five were in their confusing versions, whilst the other five contained clean versions of the code snippets (i.e.,

without the confusing constructs and idioms). The participants should provide the expected outcomes of the code snippets. As discussed, we collected information about *correctness* (whether the participant correctly predicted the program's output) and *time*. Table 3 and Figure 2 summarize the results of the survey w.r.t. correctness.

Considering Table 3, the clean versions of six code snippets present at least a 15% improvement in answer correctness when compared with the confusing versions. In particular, the presence of the *Comma Operator* atom exhibits the highest impact on misunderstanding. Frequently used constructs and idioms, such as *Post Increment* and *Omitted Curly Braces* (see Section 6), also result in many mistakes. The boxplot of Figure 2 shows a non-negligible decrease in the average number of incorrect answers when observing the clean versions of the code snippets. Also, the sample of responses with no atoms had almost no dispersion, which supports the argument that the non-confusing versions of the code snippets are easier to evaluate correctly.

Table 3: Summary of the correctness analysis

Atom	Confusing	Clean	Δ(%)
Comma Operator	28	65	+132
Automatic Semicolon Insertion	32	68	+112
Post-Increment	48	64	+33
Omitted Curly Braces	47	58	+23
Assignment as Value	56	68	+21
Implicit Predicate	58	68	+17
Logic as Control Flow	41	48	+17
Conditional Operator	60	66	+10
Pre-Increment	50	53	+6
Arithmetic as Logic	64	63	-2

Statistical analysis

We first use the *Pearson's Chi-squared test* to investigate if there is a statistically significant difference in the frequency of correct and incorrect answers—due to the versions of the code snippets (confusing and clean code). The p-values for these tests are reported in the "Chi-square test" column of Table 5. The results indicate that for five atom candidates (Comma Operator, Automatic Semicolon Insertion, Post-Increment, Assignment as Value, Implicit Predicate) the confusing versions of the code snippets have a negative impact on code understanding (p-value < 0.05). This result holds even after applying the Benjamini-Hochberg correction with a false discovery rate of 5%.

We measure the effect size of the clean version of the code snippets into the answers' correctness using the *Odds Ratio* (OR). The results are reported in the "Odds Ratio" column of Table 5. In the table, we also report the confidence interval (CI) for the OR in the "CI" column. Although many of the intervals are wide, for six of the atom candidates the lower bound of the confidence interval is greater than or equal to 1. This

indicates that, at a 95% confidence level, a developer is likely to commit less mistakes when using the clean versions. Using the thresholds established by Chen and colleagues [17] (OR = 1.68 small, 3.47 medium, and 6.71 large), for four of the atoms, Comma Operator, Automatic Semicolon Insertion, Assignment as Value, and Implicit Predicate, effect size can be considered large. For Implicit Predicate, it is medium. For instance, when comparing clean and confusing code snippets pertaining to the Comma Operator atom candidate, we observed an *Odds Ratio* of 19.02. This means that the odds of a correct answer are 19.02 times higher when interpreting the clean version—in comparison with the corresponding confusing version of the code snippet. Furthermore, with a 95% confidence level, the odds ratio is between 6.6 and 68.22, i.e., although the error margin is wide, it is strongly in favor of the clean version. If there is a significant likelihood of a developer committing a mistake when analyzing a clean version, the lower bound of the CI should be a number between 0 and 1 (since the OR is a ratio). This is the case for the four atom candidates in the lower part of the table. In particular, in the last row (Arithmetic as Logic), the OR between 0 and 1 indicates that a participant was actually more likely to misunderstand the clean version.

Finally, we also use a *Binomial Generalized Logistic Regression* analysis to investigate if either *participant education* or *participant experience* impact correctness. The findings suggest that *participant experience* impacts on the results related to correctness (Figure 3). Even though the presence of atoms of confusion reduces the number of correct answers to all *experience groups*, this effect is not uniform. For instance, novice developers (under one year of experience) provide a higher number of wrong answers for the confusing version of the code (57.8%), while developers with more than four years of experience provide more than 70% of

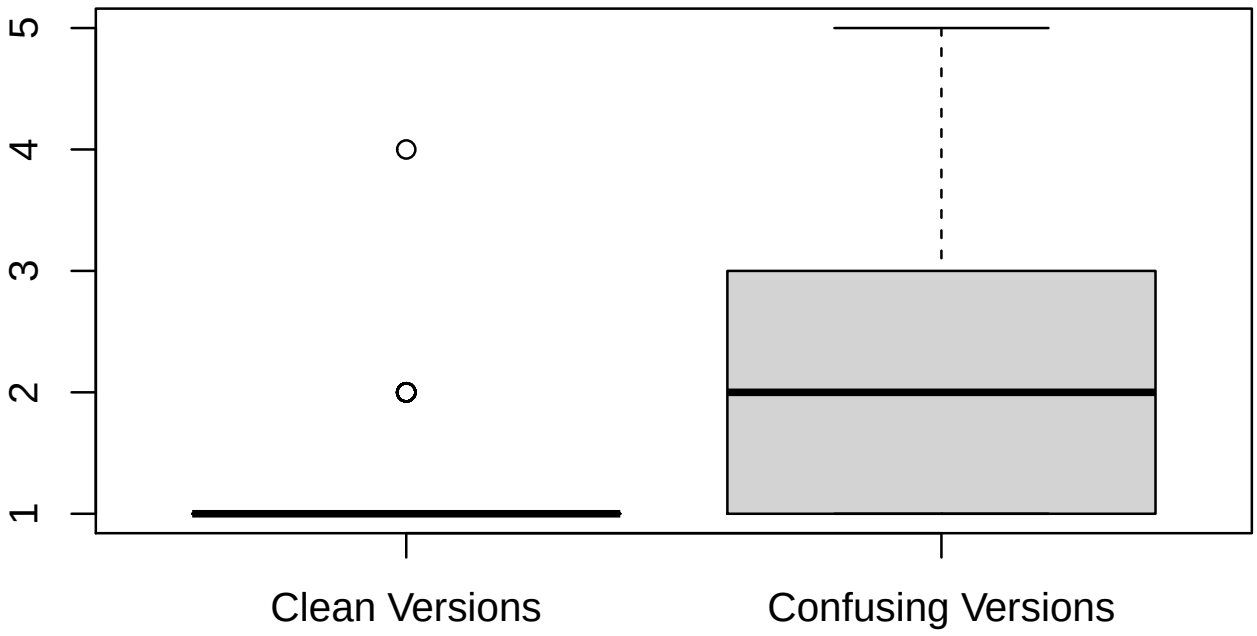

Figure 2: Number of wrong answers of each subject.

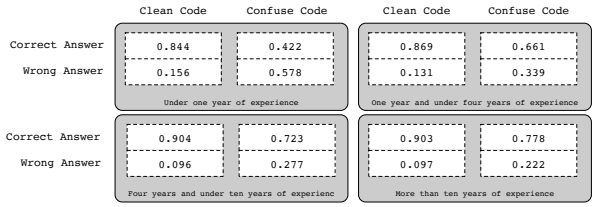

Figure 3: Impact of atoms of confusion on the correctness (over the four participant experience groups)

correct answers even when evaluating the confusing version of the code snippets. It is also important to note that, independently of the *experience group*, developers tend to provide more than 84% of correct answers while predicting the outputs for the clean versions of the code. We replicate the *Pearson's Chi-squared test* for each experience group and find that the statistical difference is less significant for those developers with more than ten years of experience. Differently from *participant experience*, the factor *participant education* does not significantly change the correctness of the answers.

Comparing to previous research, three atoms of confusion that Gopstein et al. [3] confirmed for C and C++ do not lead to a statistically significant impact on correctness in our study: Logic as Control Flow, Conditional Operator, and Pre-Increment. Our results support the Gopstein et al. findings for the remaining atoms. The work of Langhout and Aniche [6] investigates six atom candidates for Java that we also explore here for JavaScript. Among these atom candidates, we achieve similar conclusions to their study for four atom candidates: Post Increment, Conditional Operator, Arithmetic as Logic, and Omitted Curly Braces—both studies confirmed only the former as an atom of confusion. Our findings diverged from the work of Langhout and Aniche for two atoms: Pre Increment and Logic as Control Flow.

4.2. Time Analysis

Exploratory Data Analysis

Table 4 shows the average time necessary for the participants to give a correct answer about the expected outcomes of a code snippet, considering both confusing and clean versions. We do not consider cases where the participants made mistakes. Seven atom candidates required more time from participants to predict a correct response. For these seven atom candidates, developers take at least 5.90% less time on average to find a correct answer when considering the clean version of a code snippet. In the extreme case (atom candidate Comma Operator), the participants take 76.23% less time on average to find the correct answer for the clean version of the code snippets. Not all atom candidates, though, require less time for the participants to predict the answer. In fact, for the Arithmetic as Logic and the Pre-Increment atoms, the time to give a correct answer increased 29.06% and 38.19% for the clean versions—we did not confirm these atoms as atoms of confusion in the previous section.

Table 4: Time in seconds to submit a correct answer

Atom	Confusing	Clean	Δ(%)
Comma Operator	87.67	20.84	-76.23
Automatic Semicolon Insertion	46.08	22.04	-52.17
Post-Increment	28.70	25.67	-10.56
Omitted Curly Braces	48.85	30.00	-38.58
Assignment as Value	52.47	48.95	-6.71
Implicit Predicate	36.24	24.01	-33.75
Logic as Control Flow	108.94	51.07	-53.12
Conditional Operator	41.80	39.34	-5.90
Pre-Increment	30.71	42.45	38.19
Arithmetic as Logic	28.82	37.20	29.06

Statistical analysis

We use the non-parametric *Mann-Whitney U test*, also known as Wilcoxon rank-sum test, to investigate the null hypothesis that developers spend the same amount of time to correctly predict the outcome of clean and confusing versions of a code snippet. The results of Table 4 suggest that we should refute the null hypotheses for five atom candidates (Comma Operator, Implicit Predicate, Logic as Control Flow, Pre-Increment, and Arithmetic as Logic), after applying the Benjamini-Hochberg correction with a false discovery rate of 5%. For the first three, the analysis suggests that the participants need more time to predict the outcome of the code snippet in the confusing code version. For the atom candidates Arithmetic as Logic and Pre-Increment, the participants take less time to predict the outcome of the code snippets in the confusing version. We also computed the effect size using Cliff's Delta statistic. In the table, negative effect sizes suggest that it took less time to correctly predict the output of clean versions of the snippets. Considering the thresholds established by Romano et al. [18] ($0.147 < |d| < 0.33$ small, $|d| < 0.474$ medium, otherwise large), we found a large effect for the Comma Operator atom candidate; a medium effect size for Logic as Control Flow, and small effect sizes for Automatic Semicolon Insertion, Implicit Predicate, Arithmetic as Logic, Post-, and Pre-Increment.

5. Results of the Interview Study

In this section we present the results of the interviews with the practitioners. We contrast the findings of the survey presented in Section 4 here with the opinion of software developers about their preferences regarding the confusing or clean versions of the code snippets we used in our experiment.

A total of 15 practitioners took part of the interviews. We collected information regarding programming experience, familiarity with JavaScript, and their opinion about the 10 atom candidates we explored in our research. We first presented them pairs of clean and confusing code snippets and then asked them to discuss

Table 5: Hypotheses Testing (misunderstanding and time). Asterisks (*) indicate a statistically significant difference.

Atom	Chi-square test	Odds Ratio (misunderstanding)	CI	Mann-Whitey U test (workload)	Cliff's D
Comma Operator	1.1727e-10*	19.02	(6.60, 68.22)	5.8249e-08*	-0.53
Automatic Semicolon Insertion	5.8352e-11*	39.33	(9.21, 356.62)	0.09802	-0.16
Post-Increment	0.0015279*	4.83	(1.73, 15.74)	0.12006	0.15
Omitted Curly Braces	0.050962	2.35	(1.00, 5.77)	0.92529	-0.01
Assignment as Value	0.0034775*	8.39	(1.81, 79.12)	0.82681	0.02
Implicit Predicate	0.01123*	6.95	(1.46, 66.56)	0.0029039*	-0.29
Logic as Control Flow	0.292	1.54	(0.73, 3.28)	7.5862e-05*	-0.39
Conditional Operator	0.15896	2.73	(0.74, 12.57)	0.24073	-0.12
Pre-Increment	0.70147	1.25	(0.55, 2.85)	0.0062981*	0.27
Arithmetic as Logic	1	0.84	(0.22, 3.12)	0.01723*	0.23

their preferences towards one version. We did not indicate in any way whether any version was assumed to be confusing or not.

The Participants' perceptions of the atom candidates

Supporting the results of the survey, Table 6 shows that for eight out of the 10 scenarios surveyed, the majority of the respondents prefer the version of the code without the atom candidate. In no case the *neutral* ratio was higher than the option for the clean version. Only for the Conditional Operator atom candidate the participants prefer the confusing version, instead of the clean one. Figure 4 shows the code snippets for the corresponding confusing and clean versions. For this atom candidate, some participants who preferred the left-hand side version (confusing) still believed that the right-hand side version (clean) was more readable. The following quotes were extracted from the transcripts with two interviewees:

> *"I prefer to write [code using the left-hand side version], but I think [the right-hand side version] is easier to read, especially for newer programmers".*

> *"When I am programming, I write code with the conditional operator, [...], but, to be honest, I still think that [the code using the right-hand side version] is easier to understand".*

The Pre-Increment atom candidate also caused such divide in one of the interviewees, who regarded the clean version as simpler to understand, but would still opt to write code with the atom. In contrast, one of the participants found the version with the atom candidate more elegant, but recognized it was less readable, and was willing to sacrifice elegance for readability.

Table 6: Summary of participants' preferences for code snippets *with* and *without* atom candidates. The participants were only presented the code snippets, without any indication about whether one was confusing or not. Participants were also allowed to choose Neutral when they thought both sides were equally readable.

	Preference (%)		
Atom	Confusing	Clean	Neutral
Comma Operator	0	100	0
Automatic Semicolon Insertion	0	80	0
Post-Increment	20	73.33	6.67
Omitted Curly Braces	0	100	0
Assignment as Value	20	60	20
Implicit Predicate	20	73.33	6.67
Logic as Control Flow	20	60	20
Conditional Operator	60	26.67	13.3
Pre-Increment	40	46.67	13.33
Arithmetic as Logic	0	93.33	6.67
OVERALL	18	71.33	8.64

When analyzing the Logic as Control Flow atom candidate, one of the interviewees gave an example of personal experience that might motivate one to avoid writing code using this construct:

> *"This one is interesting, because I have written code that looks like the left-hand side [confusing version], and my colleagues complained that it was difficult to understand. Nowadays I prefer to write code using the version on the right-hand side [clean version]".*

For two of the atom candidates, the interviewees were unanimous in their preference for the clean version: Comma Operator and Omitted Curly Braces. Regarding the first one, we could often notice during the interviews that the *left-hand side* (with the atom of confusion) caused significant confusion among the participants. One remark about the *Comma Operator* atom of confusion is listed below:

> *"The code in [the left-hand side version] is unlikely to be understood unless the programmer knows C or C++".*

As for the atom candidate Omitted Curly Braces, one of the interviewees mentioned that, although they understand why one would opt not to use braces for simple if-then-else statements, they still advised against it, on grounds that:

```
let config = {size: 3, isActive: false};
const_config = config.isActive === true
             ? config
             : {size: 10};
console.log(_config.size);
```

Listing 2: *Left-hand side* (using the *Conditional Operator* atom).

```
let config = {size: 3, isActive: false}
let _config;
if(config.isActive === true) {
  _config = config;
}
else{
   _config = {size: 10};
}
console.log(_config.size);
```

Listing 3: *Right-hand side* (without the atom).

Figure 4: Example of a pair of code snippets used in the interview pertaining to the *Conditional Operator* atom candidate

> *"[I prefer the right-hand side version of the code . . .] If I want to see well-written, easily understandable code, then I also have to do my job. Therefore I believe that, since I do not know who is on the other end maintaining this code, and it could be any person with any level of expertise, then I try to write readable, easy-to-understand code".*

Confusion in JavaScript Code

The final remarks drawn from the interviews are related to potentially confusing constructs suggested by the participants and their perspectives on JavaScript as a language. From the latter, we can also uncover some other forms in which the language itself might contribute to writing confusing code.

One of the participants mentioned that the use of JavaScript's prototype-based inheritance can make it difficult to understand code, particularly when involving deep prototype chains. When asked about particular JavaScript constructs or patterns that can make code difficult to understand, three participants cited the callback pattern—potentially leading to several levels of nested function calls—as extremely difficult to assimilate. One of the respondents stated:

> *"Nested callbacks are very confusing. Even writing them can be confusing, let alone understanding them."*

Two other developers implicitly touched upon the callback pattern, mentioning that it can be difficult to understand *asynchronous* programming in JavaScript. We also found other idioms that are JavaScript atom candidates, including *property access* via array subscription (`V1['P1']` instead of `V1.P1`) and *arrow functions* (see listings 4 and 5). As future work, we aim at investigating whether or not these atom candidates are more likely to introduce confusion in JavaScript code.

```
let inc = (x) => x + 1;
```

Listing 4: Example of arrow function.

```
let inc = function(x) {
  return x + 1
}
```

Listing 5: Alternative version without arrow function.

6. Results of the MSR Effort

In this section we present the results of our mining software repository effort, whose goal is to reveal how prevalent atom candidates are in open source JavaScript projects. We mined 72 open source JavaScript repositories to understand how often atoms of confusion arise in real software. Similarly to previous studies [7, 5], which investigate the prevalence of atoms of confusion in open source C and C++ projects, we found that atoms of confusion frequently arise in JavaScript open source systems. For instance, the six most frequently found atoms occur in at least 80% of the projects. Considering the extremes, atom candidates Conditional Operator and Implicit Predicate were found in all repositories we mined, while Comma Operator occurred in only 15% of them (as seen in Table 7).

Table 7: Summary of atom candidate occurrences in our dataset.

Atom	Projects	Occurr./KLOC
Implicit Predicate	100%	19.89
Conditional Operator	100%	10.16
Omitted Curly Braces	91.67%	6.61
Post Increment	90.28%	5.62
Pre Increment	83.33%	1.04
Assignment as Value	81.94%	1.02
Logic as Control Flow	56.94%	0.95
Automatic Semicolon Insertion	33.33%	0.13
Arithmetic as Logic	23.61%	0.02
Comma Operator	15.28%	0.03

Considering all JavaScript projects in our dataset, we found a total of 364 873 atom candidates, though four atoms are responsible for 92.97% of this total: Implicit Predicate, Post Increment, Conditional Operator, and Omitted Curly Braces. The first two, based on the results of Section 4, can be considered atoms of confusion. The remaining atom candidates comprise 25 621 occurrences combined, 7.03% of the total number

of occurrences. The Comma Operator and Arithmetic as Logic atoms are the ones that arise less frequently (0.001% in total with 232 and 165 occurrences, respectively).

We calculated the frequency of each atom per KLOC. The four atoms mentioned in the previous paragraph, Implicit Predicate, Post Increment, Conditional Operator, and Omitted Curly Braces, occur frequently in the analyzed projects. All of them have more than 5 occurrences per KLOC on average. In particular, two atom candidates occur very frequently, Implicit Predicate (19.89 occurrences per KLOC) and Conditional Operator (10.16 occurrences per KLOC). On the other hand, Arithmetic as Logic and Comma Operator occur less than once for every 50 KLOC.

The results of our survey and interviews suggest that the Conditional Operator does not contribute significantly as a source of misunderstanding (increasing the number of wrong answers in 9% of the cases, according to our survey). Nonetheless, the Post-Increment Expression and Automatic Semicolon Insertion atoms are listed in the top four sources of misunderstanding (Table 3), and also frequently appear in open source systems. As such, refactoring existing systems to avoid Post-Increment and Automatic Semicolon Insertion might improve the readabiliy of large amounts of code in JavaScript projects.

7. Discussion

Our work has several implications. First, it adds external validity to the work of Gopstein et al. [3], which investigates the impact of atom candidates on understanding C,C++ code. That is, similarly to their work, the atom candidates Comma Operator, Post/Pre Increment, Omitted Curly Braces, Assignment as Value, Implicit Predicate, and Logic as Control Flow seem to make JavaScript code harder to understand. For five of them, the difference is statistically significant, with a large effect size for four atoms. Our results also refute the hypothesis that Arithmetic as Logic is an atom of confusion (i.e., a source of misunderstanding). In comparison to the original work of Gopstein et al. [3], our study led to some differences in the effect size of the atom candidates. Altogether, we answer our first research question.

> **Answer to RQ1:** The first study (survey) gives evidence that some of the atom candidates in C and C++ programs that also exist in JavaScript correspond to a source of misunderstanding in JavaScript code.

The results of the interview study complement the understanding of atoms of confusion because the participants make clear the existence of a trade-off between code comprehension and other quality attributes. For instance, most of the participants prefer the version of the code with the Conditional Operator, even though they agree that its use might contribute to the misunderstanding of JavaScript code, particularly when novices are maintaining the codebase. The participants of the interview study also mentioned other possible sources of misunderstanding in JavaScript, including the use of prototype-based inheritance and nested callbacks (as discussed in Section 5). Other JavaScript atom candidates include Object Destructuring, Array

Spread, Object Spread, and Type Conversion. In summary, the results of the second study (interviews) allow us to answer the second research question.

> **Answer to RQ2:** The qualitative analysis of the interviews supports the results of the first study, since JavaScript developers most often agree that atoms of confusion compromise source code understanding.

The results of the third study (mining open source JavaScript repositories) provides evidence that, although atoms of confusion compromise program comprehension, they frequently appear in open source JavaScript projects. In particular, seven, out of 10 atoms considered in our study, appear in more than 50% of the 72 projects we analyzed. Furthermore, at least two of them are used intensively, more than once for every 200 lines of code. In summary, the third study allows us to answer the third research question.

> **Answer to RQ3:** The MSR study reveals that the several atom candidates explored in our research appear frequently in practice, and cleaning up the use of Post-Increment/Decrement and the Automatic Semicolon Insertion might improve the readabiliy of JavaScript code substantially.

8. Threats to Validity

Conclusion validity. In the context of our survey study, we apply different non-parametric statistical tests appropriate for the cases where data was categorical (correctness, Chi-square test of independence) and continuous (time, Mann-Whitney U test). Furthermore, besides reporting p-values, we also report effect size measures appropriate for each scenario (odds-ratio and Cliff's delta) and apply a p-value correction technique to avoid the multiple testing problem. Finally, it could be argued that the size of the samples is insufficient to make conclusions for some of the atoms, a common problem in Empirical Software Engineering. We estimate the sample size for each one of the atom candidates, considering that each one had a different number of samples (Table 3). To that end, we use the ϕ measure of effect size for each atom (based on the Chi-square statistic), the standard α coefficient of 0.05, and set the expected statistical power to 0.8, as usually employed in the literature [19]. We find out that the sample size is sufficient for all but one of the atoms where we had a statistically significant result for correctness, Implicit Predicate.

Construct validity. We use correctness as a proxy for program comprehension and predicted outcomes of small programs as a measure of correctness. As discussed elsewhere [20], this approach is a test of the developers' ability to trace programs. Although this is a common approach in studies about atoms of confusion [11, 6, 3], other measures of correctness could have been employed, potentially yielding different results. As a complement to correctness in the survey, we have measured the time it took for the participants to correctly predict the outcome of the code snippets (either with or without atom candidates). Furthermore, we have asked the interviewees about their preferences when comparing confusing and clean versions of the programs.

Internal validity. Since the survey was conducted online with unknown participants, we have no way of confirming their levels of education and experience. We mitigate this threat by, in the data analysis, explicitly accounting for programmer experience and using an experimental design that allows us to isolate the impact of the treatment from factors such as experience and formal education level. Also, we did not have a way to prevent respondents from cheating, such as running the code on an interpreter, or consulting other people. We partially mitigate this threat by presenting images with source code, instead of text. This creates an obstacle for participants to run the code while taking the survey.

External validity. Our results suggest that the selected idioms and code constructs may lead to confusion for small code snippets, but it is not clear if that result extrapolates to other scenarios. Even though it is likely that in larger code bases the confusion induced by these constructs may be even greater, the existence of additional context may mitigate this effect. Another possible threat to the generalizability of our results, in particular for the survey and interviews, lies in the fact that the analyzed atoms may rarely occur in real JavaScript code bases. To mitigate that threat, we have analyzed 72 popular open source JavaScript repositories and found out that most of them are common, occurring at least once per 1,000 lines of code. Only Arithmetic as Logic and Comma Operator occur less frequently than once per 10,000 lines of code.

9. Conclusions

This paper reports the results of a mixed-method research effort that allowed us to better understand the impact of atoms of confusion in JavaScript code. First, we conducted a survey with 140 JavaScript developers, asking them to predict the output of some code snippets (with and without atoms). We provide evidence that five atom candidates significantly impact program comprehension activities and can be considered atoms of confusion. Our efforts have two implications for practice. First, we present evidence that not all atoms of confusion validated to statically typed languages (such as C, C++, or Java) led to a statistically significant impact on program understanding for JavaScript (a dynamically typed language). Besides that, our findings might be used to alert developers to avoid writing JavaScript code with certain atoms of confusion (e.g., Comma Operator, Automatic Semicolon Insertion, Post Increment/Decrement, Assignment as Value, and Implicit Predicate). Finally, our results might help tool developers to create program transformation tools for removing atoms that frequently appear in software. Our research also pointed out additional JavaScript constructs and idioms that might introduce misunderstanding. These include nested callbacks, *property access*, and *arrow functions*. As future work, we intend to reproduce our survey to validate whether or not these additional *atom candidates* truly affect the understanding of JavaScript code.

References

[1] F. Hermans, The Programmer's Brain: What every programmer needs to know about cognition, Manning, 2021.

[2] S. Ajami, Y. Woodbridge, D. G. Feitelson, Syntax, predicates, idioms: what really affects code complexity?, in: Proceedings of the 25th International Conference on Program Comprehension, ICPC, IEEE Computer Society, Buenos Aires, Argentina, 2017, pp. 66–76.

[3] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K. Yeh, J. Cappos, Understanding misunderstandings in source code, in: ESEC/SIGSOFT FSE, ACM, https://doi.org/10.1145/3106237.3106264, 2017, pp. 129–139.

[4] F. Castor, Identifying confusing code in swift programs, In Proceedings of the VI CBSoft Workshop on Visualization, Evolution, and Maintenance 1 (6) (2018) 1–8.

[5] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, R. Gheyi, An investigation of misunderstanding code patterns in C open-source software projects, Empirical Software Engineering 24 (4) (2019) 1693–1726.

[6] C. Langhout, M. Aniche, Atoms of confusion in java, in: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension, ICPC '21, New York, NY, USA, 2021, to appear.

[7] D. Gopstein, H. H. Zhou, P. G. Frankl, J. Cappos, Prevalence of confusing code in software projects: atoms of confusion in the wild, in: MSR, ACM, https://doi.org/10.1145/3196398.3196432, 2018, pp. 281–291.

[8] E. George, W. G. Hunter, J. S. Hunter, Statistics for experimenters: Design, innovation, and discovery, Wiley, 2005.

[9] S. R. Tilley, D. B. Smith, S. Paul, Towards a framework for program understanding, in: WPC, IEEE Computer Society, DOI: 10.1109/WPC.1996.501117, 1996, p. 19.

[10] A. B. O'Hare, E. W. Troan, Re-analyzer: From source code to structured analysis, IBM Systems Journal 33 (1) (1994) 110–130.

[11] B. de Oliveira, M. Ribeiro, J. A. S. da Costa, R. Gheyi, G. Amaral, R. de Mello, A. Oliveira, A. Garcia, R. Bonifácio, B. Fonseca, Atoms of confusion: The eyes do not lie, in: Proceedings of the 34th Brazilian Symposium on Software Engineering, SBES '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 243–252. doi:10.1145/3422392.3422437.
URL `https://doi.org/10.1145/3422392.3422437`

[12] D. Gopstein, A.-L. Fayard, S. Apel, J. Cappos, Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion, ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 605–616. doi:10.1145/3368089.3409714.
URL https://doi.org/10.1145/3368089.3409714

[13] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszédes, R. Ferenc, A. Mesbah, Bugsjs: A benchmark of javascript bugs, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), IEEE, 2019, pp. 90–101.

[14] E. D. Canedo, R. Bonifácio, M. V. Okimoto, A. Serebrenik, G. Pinto, E. Monteiro, Work practices and perceptions from women core developers in OSS communities, in: ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020, ACM, 2020, pp. 26:1–26:11. doi:10.1145/3382494.3410682.
URL https://doi.org/10.1145/3382494.3410682

[15] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, J. Tibble, .ql: Object-oriented queries made easy, in: R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers, Vol. 5235 of Lecture Notes in Computer Science, Springer, 2007, pp. 78–133. doi:10.1007/978-3-540-88643-3_3.
URL https://doi.org/10.1007/978-3-540-88643-3_3

[16] O. Rodriguez-Prieto, F. Ortin, An efficient and scalable platform for java source code analysis using overlaid graph representations (support material website) (2020).

[17] H. Chen, P. Cohen, S. Chen, How big is a big odds ratio? interpreting the magnitudes of odds ratios in epidemiological studies, Communications in Statistics - Simulation and Computation 39 (4) (2010) 860–864.

[18] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?, in: Proceedings of the Annual meeting of the Florida Association of Institutional Research, 2006, pp. 1–3.

[19] P. D. Ellis, The Essential Guide to Effect Sizes: An Introduction to Statistical Power, Meta-Analysis and the Interpretation of Research Results., Cambridge University Press, 2010.

[20] D. Oliveira, R. Bruno, F. Madeiral, F. Castor, Evaluating code readability and legibility: An examination of human-centric studies, in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, IEEE, Adelaide, Australia, 2020, pp. 348–359.