Dazed and Confused: Studying the Prevalence of Atoms of Confusion in Long-Lived Java Libraries

Wendell Mendes, Oton Pinheiro, Emanuele Santos, Lincoln Rocha and Windson Viana
Department of Computing, Federal University of Ceará, Fortaleza, Ceará, Brazil
wendell.mendes@sti.ufc.br, otonpneto@alu.ufc.br, emanuele@dc.ufc.br, lincoln@dc.ufc.br, windson@virtual.ufc.br

***Abstract*—Program comprehension is a fundamental activity in software maintenance and evolution, impacting several tasks such as bug fixing, code reuse, and implementation of new features. The Atom of Confusion (AC) is considered the smallest piece of code that can confuse programmers, difficulting the correct understanding of the source code under consideration. Previous studies have shown that these atoms can significantly impact the presence of bugs in C++ projects and increase the time and effort to code understanding in C++ and Java programs. To gather more evidence about the diffusion of ACs in the Java ecosystem, we conduct a study to analyze the prevalence, co-occurrences (at the class level), and evolution of ACs in 27 long-lived Java libraries. To support our investigation, we developed an ACs automatic search tool, which found 11.404 occurrences in the studied libraries. The *Conditional Operator* and *Logic as Control Flow* ACs were the most prevalent among the 10 types of ACs assessed. Our findings show that *Conditional Operator* and *Logic as Control Flow* were more likely to co-occur in the same class. Finally, we observed that the prevalence of ACs did not decrease over time. On the contrary, in 13 libraries, the presence grew proportionally more than the size of the library in lines of code. Furthermore, in 15 libraries, the fraction of Java classes containing at least one AC also increases over time.**

***Index Terms*—Empirical Study, Program Comprehension, Atoms of Confusion, Long-lived Java Projects**

I. Introduction

Code comprehension is the activity in which software engineers seek to understand a computer program having its source code as the main reference [2]. Understanding source code is critical in software development, both in creating new features and in maintaining existing ones. Increasing knowledge about the code helps software engineers to better perform maintenance activities such as fixing *bugs*, code refactoring, code reusing, and even documentation writing [23].

In software development, developers frequently deal with code snippets that were not initially written by themselves. The developers' cognitive process involves identifying, understanding, and analyzing code written by other developers. It is not rare the cases in which, for a particular code snippet, the human understanding diverges from the machine's interpretation, leading to an erroneous conclusion about the code snippet's outcome in a future execution [9].

Previous studies showed that code comprehension is the most dominant activity in the development process, consuming about 58% of the total time spent [19, 25]. Rahman in [22] observed that when programmers are involved in high comprehension effort, they navigate and make edits at a significantly slower rate. Ebert et al. in [7] observed that code reviewers often do not understand the change being reviewed or its context [7]. In this circumstance, confusing code impacts code comprehension and, hence, the development process. Developers tend to understand certain code structures more quickly than other ones (more challenging), e.g., `for` loops take more time to be understood than sequences of `if` [1]. Some programming practices also affect the code readability [6].

Gopstein et al. in [9] identified patterns in code that are responsible for creating confusion in developers. These patterns have been called **Atoms of Confusion (ACs)**. The authors conducted two surveys to assess the impact of these confusing patterns. First, code snippets written in the C language were analyzed, with and without the presence of ACs, comparing the correctness of their execution results. The researchers concluded that codes containing ACs make understanding more complex when compared to codes with equivalent functions that do not include these atoms. This seminal work showed that the presence of these patterns could significantly impact correctness and the time and effort for program understanding. Despite the importance shown by Gopstein et al., AC studies are still few. The impact caused by ACs in other programming languages, such as Java, and how this phenomenon is diffused on long-lived software systems calls for further investigation.

Inspired by the study of Langhout and Aniche [15], we examined the prevalence of ACs in the Java ecosystem. Firstly, we developed a tool for searching ACs in Java source code on top of Spoon[1], a Java source code analysis and transformation tool. Next, we investigated the (1) prevalence, (2) co-occurrence, and (3) evolution of ACs in 27 well-known and widely adopted long-lived Java libraries from open-source ecosystems (21 libraries from Apache Software Foundation plus Gson, Hamcrest, Jsoup, JUnit, Mockito, and X-Stream).

We detected the prevalence of $11,404$ ACs in $449,885$ lines of code analyzed. Our tool found 9 types of ACs in these long-lived Java libraries. Apache libraries like Math and Compress had nine types of ACs, while Gson library had seven types. We detected the *Logic as Control Flow* and the *Conditional Operator* atoms in all studied libraries. On the other hand, the *Arithmetic as Logic* and *Repurposed Variables* ACs appeared only in 3 of them, and we did not find the *Omitted Curly Braces* in any library. Our findings shows that *Conditional Operator* and *Logic as Control Flow* ACs are more likely to co-occur in the same class. The evolution analysis showed the

[1]https://spoon.gforge.inria.fr/

presence of ACs occurred since the first version of analyzed projects. The absolute number of ACs has increased in all studied projects except for JUnit. Thus, we were able to observe that the number of ACs grows proportionately (or more highly) than the size of the libraries in lines of code.

The remainder of this work is organized as follows. Section II discuss the related work. The methodology adopted in this study is described in Section III. In Section IV we present the tool we built to support our study. Next, in Section V, we describe the study results and, in Section VI, we present further results discussion. Section VII describes the threats to the validity of our study. Finally, Section VIII presents the final considerations and proposals for further investigation.

II. Related Work

Gopstein et al. [9] introduced the concept of **Atom of Confusion** (AC), in which an AC can be defined as the smallest piece of code capable of causing confusion in developers, causing erroneous conclusions about their behavior. The hypothesis is that the existence of these atoms affects the source code understanding and may lead developers to make mistakes in maintenance tasks, introducing bugs in the system.

To assist researchers interested in the study of atoms of confusion, Castor [3] defined an AC as a code pattern that is: (i) precisely identified; (ii) likely to confuse; (iii) replaceable by a functionally equivalent code pattern that is less likely to confuse; and (iv) indivisible.

In [9], Gopstein et al. pointed out 15 ACs that cause significant confusion when present in source code. In complementary work, Gopstein et al. [10] observed a strong relationship between lines of code containing ACs and the occurrence of *bugs*, showing that *bug-fix commits* removed more ACs when compared to other commit types. Similarly, they observed that the lines containing ACs caused more confusion, as these pieces of code tended to be more commented than others.

In a more recent work, Gopstein et al. [11] conducted a qualitative study that noted that quantitative studies may be underestimating the amount of misunderstanding that occurs during the studies assessments, since correct answers on assessment tasks do not guarantee that there was no confusion in the process of understanding source code.

de Oliveira et al. [5] evaluated code interpretations with and without ACs using an eye tracker. From an aggregate perspective, a 43.02% increase in time and a 36.8% increase in gaze transitions were observed in code snippets with ACs. The authors also observed that the regions that received the most eye attention were the regions containing ACs.

Based on the study of Gopstein et al. [9], Langhout and Aniche [15] defined atoms of confusion in the context of the Java programming language. This study analyzed and translated the 19 ACs of confusion defined by Gopstein et al., resulting in a list of 14 reproducible ACs in Java. They also evaluated the perceptions and impacts of ACs on novice developers. The results showed developers are 4.6 to 56 times more likely to make misunderstandings in 7 of the 14 ACs studied. Furthermore, when the authors confronted the study participants with two versions of code, with and without atoms of confusion, they reported that the version containing ACs is more confusing and less readable in 10 of the 14 ACs investigated. Thus, the study shows that these ACs can confuse novice developers [15]. Table I presents the list of the ten ACs based on that work, their respective Java translations, and the code with the confusion removed.

Following the study of Langhout and Aniche [15] in the context of Java ecosystem, our research proposes investigating the incidence of ACs using an automated search tool. With the development and validation of this tool, we could more quickly observe the prevalence, the co-occurrence, and the ACs evolution in Java projects. Our findings and implications are discussed in the following sections.

III. Methodology

A. Study Design

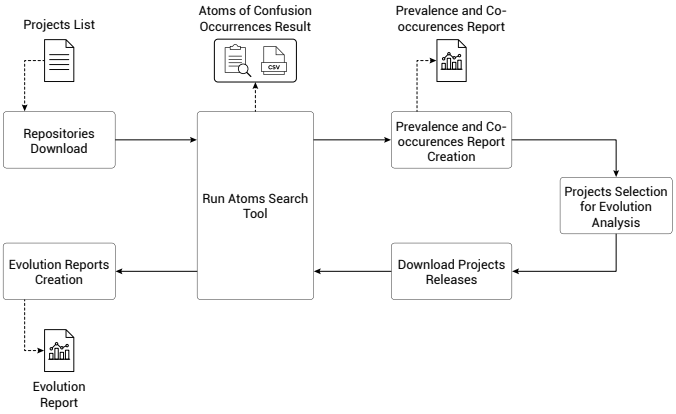

Fig. 1. Prevalence and Evolution study workflow

Our study investigates the prevalence of Atoms of Confusion in open-source long-lived Java libraries. The goal is to quantify to what extent ACs are prevalent in Java libraries and make this information available to researchers and practitioners as the first step for further investigation concerning causality issues and how to properly address this phenomenon.

Figure 1 shows our study's main phases. First, we selected 27 long-lived Java libraries (see Section III-C) to serve as subjects in our empirical evaluation. We also investigated in those projects the co-occurrence of ACs. Finally, we selected 24 projects to investigate the evolution of the prevalence of ACs over time. We did not consider three projects since they did not have a sufficient number of versions for analysis.

B. Research Questions

The research questions we investigated in the study were:

RQ1. *What is the prevalence of atoms of confusion in long-lived Java libraries*?

The purpose is to provide a first insight into the prevalence of ACs. We checked their occurrences in the 27 selected

TABLE I
ATOMS OF CONFUSION IN JAVA ADAPTED FROM [15]

Atom of Confusion Name	Acronym	Snippet with Atom of Confusion	Snippet without Atom of Confusion
Infix Operator Precedence	IOP	`int a = 2 + 4 * 2;`	`int a = 2 + (4 * 2);`
Post-Increment/Decrement	Post-Inc/Dec	`a = b++;`	`a = b;` `b += 1;`
Pre-Increment/Decrement	Pre-Inc/Dec	`a = ++b;`	`b += 1;` `a = b;`
Conditional Operator	CO	`b = a == 3 ? 2 : 1;`	`if(a == 3){b = 2;}` `else{b = 1;}`
Arithmetic as Logic	AaL	`(a - 3) * (b - 4) != 0`	`a != 3 && b != 4`
Logic as Control Flow	LaCF	`a == ++a > 0 || ++b > 0`	`if(!(a + 1 > 0)) {b += 1;}` `a += 1`
Change of Literal Encoding	CoLE	`a = 013;`	`a = Integer.parseInt("13", 8);`
Omitted Curly Braces	OCB	`if(a) f1(); f2();`	`if(a){ f1(); } f2();`
Type Conversion	TC	`a = (int) 1.99f;`	`a = (int) Math.floor(1.99f);`

Repurposed Variables	RV	`int a[] = new int[5];`
		`a[4] = 3;`
		`while (a[4] > 0) {`
		`  a[3 - v1[4]] = a[4];`
		`  a[4] = v1[4] - 1;}`
		`System.out.println(a[1]);`
			`int a[] = new int[5];`
			`int b = 5;`
			`while (b > 0) {`
			`  a[3 - a[4]] = a[4];`
			`  b = b - 1;}`
			`System.out.println(a[1]);`

libraries. The study measured for each library the amount of ACs present and the occurrence of distinct ACs' types.

RQ2. *To what extent do different types of atoms of confusion co-occur, at the class file level, in long-lived Java libraries*?

Besides measuring the prevalence of atoms of confusion, we computed the co-occurrence of ACs in the same Java file (a class). The goal is to observe tendencies for certain types of ACs to occur together in a Java class. ACs co-occurrence may indicate similarities in ACs code snippets structures, but also if classes with ACs co-occurrences are changed by more than one developer, or, even if the role implemented by a class contributes to the presence of ACs (e.g., a class implementing mathematical operations is more likely to have ACs).

RQ3. *How long do Atoms of Confusion survive in long-lived Java libraries*?

We studied the evolution of the occurrences of the Atoms of Confusion in 24 libraries from the set of 27 selected. The goal is to evaluate how long ACs survive during the life span of a library. We observed the prevalence of ACs over time in these libraries. The analysis was made over a total of 455 versions of all 24 libraries studied.

C. Selection of Long-lived Java Libraries

We performed our analysis using the same set of long-lived Java libraries used in the study of Lima et al. [16]. This set comprises 27 libraries, 21 libraries from the Apache Commons[2] ecosystem and six well-known libraries from other ecosystems. In addition, these projects present an automatically executable test suite and Maven (or Gradle) as their build system, which helps us correctly handle them with Spoon [20].

These 27 libraries are used on thousands of systems and are long-lived Java projects over ten years old. Despite this, most have had recent releases within the last two years. These libraries are, therefore, projects that are constantly updated. Consequently, we believe that investigating the presence of AoC in these projects is a good indication of how this phenomenon occurs and evolves in Java projects. Table IV summarizes the selected libraries for this study.

[2]https://commons.apache.org/

IV. THE ACS SEARCH TOOL

To answer our research questions, we developed a tool to automatically search ACs in programs written in Java. The tool used in this work aims to check for the presence of ACs in Java classes and provides information about them. This software was developed using Spoon, an open-source library to analyze, rewrite, transform and transpile Java source code [20]. The developed tool analyzes the *.java* files in source code from software repositories. The information found can be exported in a report as a CSV file. This CSV file shows the code snippets that contain ACs, their types, class names, and the lines in which they were found.

Currently, the tool is able to detect 10 of the 13 types of ACs presented by Langhout and Aniche [15]: *Infix Operator Precedence, Post-Increment/Decrement, Pre-Increment/Decrement, Conditional Operator, Arithmetic as Logic, Logic as Control Flow, Change of Literal Encoding, Omitted Curly Braces*, *Type Conversion* and *Repurposed Variables*. The detection of *Repurposed Variables* ACs is partially covered. This atom consists of "misusing" of an existing variable for another purpose. In this sense, automatically detecting the use of a variable for another purpose is not trivial due its "semantic" evaluation. Hence, our tool covers only two of the three cases of this atom as described by Langhout [14]. All rules defined for each AC covered by the tool are presented in detail on the tool's source code repository available in the following link: https://anonymous.4open.science/r/bohr-aoc-api-3E3D/.

Langhout [14] argues that are some kind of ACs that can be more easily avoided, such as *Remove Indentation*, *Indentation*,

and *Dead, Unreachable, Repeated*. For example, developers can avoid the *Remove Indentation* and *Indentation* ACs by using automatic code formatters present in most code editors. *Dead, Unreachable, Repeated* atom is detectable by static code tools (linters) available as a plugin for most IDEs, which inform the presence of this AC through warning messages. Therefore, we decided not to include the detection of these three ACs in this version of our search tool.

A. Search Tool Dataset

To minimize bias in the automatic identification of ACs, we also manually built a double-checked gold standard dataset to assess the precision and recall of our tool. Figure 2 shows the workflow of this evaluation. The dataset creation was divided in two steps: projects selection and manual inspection.

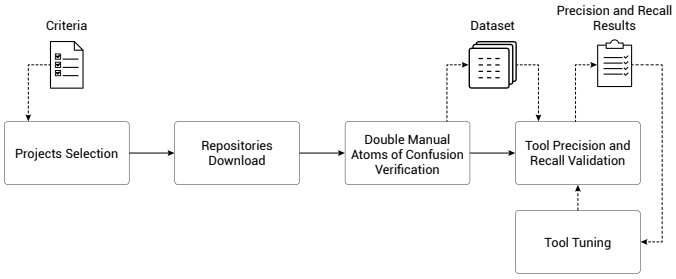

Fig. 2. Precision and Recall study workflow

To create the dataset, we selected four open-source Java projects that met the following criteria: (i) projects having more than 50% Java source code; (ii) having up to twenty thousand lines of code (LoC), excluding tests; (iii) possessing at least one thousand stars on GitHub, (iv) having commits and releases in 2021; and (v) containing at least five different types of ACs. We applied criterion (ii) to make it more feasible to check the occurrences of ACs manually and compare them with those ACs detected by our search tool. In this sense, observing the classification of Pinto et al. [21], we choose small projects (i.e., LoC $\leq$ 20,000). To check criterion (v), we ran the search tool on several candidate projects that met the previous four criteria and checked their accuracy.

Four projects satisfied the inclusion criteria: FastUtil[3], Moshi[4], Jimfs[5], and uCrop[6]. As none of these projects contained the *Repurposed Variables* and *Arithmetic as Logic* ACs, we created a sample project, by extracting Java files containing those two ACs from the Guava (version 31.0.1) and Redisson (version 3.6.16) projects. Table II presents the selected projects and their respective versions used in the evaluation.

We manually checked all Java files in these projects' main source code package, excluding the test files, to search for atoms of confusion. Two master students performed this verification independently. In this process, perspective alignment meetings were held at the end of the verification of each project to ensure both students had the same understanding concerning the occurrence of ACs. The final result was a dataset of ACs occurrences, containing the code snippet of each AC, its types, the class name in which it was found, and its location (the number of the start and end lines of code). TableIII shows the numbers of ACs and their types we found.

[3]https://github.com/vigna/fastutil
[4]https://github.com/square/moshi
[5]https://github.com/google/jimfs
[6]https://github.com/Yalantis/uCrop

TABLE II
PROJECTS USED IN THE PRECISION AND RECALL EVALUATION

	FastUtil	Moshi	Jimfs	uCrop	Sample
Version	8.5.6	1.12.0	1.2	2.2.7	-
LoC	1,622	5,783	7,823	4,309	850
#Classes	42	30	59	32	5
#Classes with AC	13	15	30	17	5
#AC	86	151	118	111	23
#AC Types	7	6	7	5	6

B. Precision and Recall Evaluation

We ran the search tool on the selected projects and compared its results with manually annotated information in the dataset to verify its precision and recall. In the first iterations, we did not identify precision problems. All the ACs the tool detected were also tagged manually. Therefore, there was no identification error. However, some issues with the tool recall appeared. For example, the tool did not was able to identify all occurrences of the *Type of Conversion* (ToC) and *Infix Operator Precedence* (IOP) atoms of confusion.

In the case of ToC, the recall issue occurred due to unsupported types of conversions (i.e., literals, unary operations, and binary operations). In the IOP case, we needed to improve the detection of the operation's parenthesis hierarchy, as well as deal with the ambiguous interpretation of '+' character, which could indicate an addition operation or a string concatenation operation. Therefore, we fixed it and added new rules in our tool to guarantee 100% of precision and correctly identify all the ACs previously found and registered in the dataset. All detection rules are described in the tool's source code repository https://anonymous.4open.science/r/bohr-aoc-api-3E3D/.

TABLE III
ACS OCCURRENCES IN PRECISION AND RECALL EVALUATION DATASET

AC Type	FastUtil	Moshi	Jimfs	uCrop	Sample
IOP	8	5	5	31	1
Pre-Inc/Dec	3	-	8	-	-
Post-Inc/Dec	20	10	4	8	-
CO	18	74	31	31	1
OCB	1	-	-	-	-
LaCF	12	65	58	28	7
AaL	-	-	-	-	4
CoLE	-	2	9	-	2
TC	24	4	3	13	3
RV	-	-	-	-	2

TABLE IV
PROJECTS INFORMATION AND PREVALENCE RESULTS

Library	Version	LoC	Classes	Classes w/ACs	ACs	Types	IOP	Pre-Inc /Dec	Post-Inc /Dec	CO	LaCF	AaL	CoLE	TC	RV
BCEL	6.5.0	31,686	391	76	322	7	9	3	14	117	145	-	1	33	-
BeanUtils	1.9.4	11,644	111	36	174	5	5	-	3	77	87	-	-	2	-
CLI	1.5.0	2,151	23	12	84	3	-	-	1	23	60	-	-	-	-
Codec	1.15	9,313	72	37	436	7	13	7	123	53	157	-	6	77	-
Collections	4.4	28,955	326	96	565	6	42	9	22	220	270	-	-	2	-
Compress	1.2.1	44,730	359	174	1,155	9	88	56	139	279	360	1	20	197	15
Configuration	2.7	28,011	260	92	342	6	2	1	5	152	181	-	-	1	-
DBCP	2.9.0	14,454	66	31	127	2	-	-	-	46	81	-	-	-	-
DbUtils	1.7	3,074	46	19	29	2	-	-	-	11	18	-	-	-	-
Digester	3.2	9,917	168	39	94	5	5	5	5	13	66	-	-	-	-
Email	1.5	2,815	23	12	50	5	-	-	1	14	32	1	-	2	-
Exec	1.3	1,757	32	11	38	4	2	-	-	7	28	-	-	1	-
FileUpload	1.4	2,425	39	7	26	5	-	1	6	1	17	-	-	1	-
Functor	1.0	5,861	158	111	495	3	18	-	-	163	314	-	-	-	-
IO	2.11.0	14,024	180	77	358	7	11	5	8	146	133	-	2	53	-
Lang	3.12.0	29,745	215	80	880	8	138	16	52	242	369	-	13	49	1
Math	3.6.1	100,364	990	390	4,174	9	2,753	34	129	616	484	5	32	119	2
Net	3.8.0	20,199	212	86	389	6	7	29	69	64	147	-	-	73	-
Pool	2.11.1	5,905	49	16	80	5	7	-	2	20	47	-	-	4	-
Proxy	1.0	2,072	43	10	15	4	1	-	1	7	6	-	-	-	-
Validator	1.7	7,619	64	41	167	5	-	1	-	37	122	-	2	5	-
Gson	2.8.9	8,342	77	33	263	7	5	3	12	142	91	-	2	8	-
Hamcrest	2.2	3,505	80	9	17	3	-	-	1	3	13	-	-	-	-
Jsoup	1.14.3	13,714	73	39	323	6	19	4	6	111	170	-	-	13	-
JUnit	5.8.2	30,977	645	133	284	4	3	-	1	100	180	-	-	-	-
Mockito	4.3.0	20,298	467	87	249	5	7	3	16	56	167	-	-	-	-
X-Stream	1.4.19	21,859	361	164	507	6	16	9	16	231	218	-	-	17	-

V. RESULTS

Once the effectiveness of our tool was confirmed, we started the study on the prevalence of ACs in long-lived Java libraries.

A. RQ1. What is the prevalence of atoms of confusion in long-lived Java libraries?

To provide a first insight into the prevalence of ACs, we computed the number of ACs present in each library and the amount of ACs per type. Table IV shows the results for all selected libraries, the number of classes containing ACs, the number of ACs found, and the number of types present in each library. $11,404$ ACs were found, with an average ≈ 2.29 ACs per class and a rate ≈ 1.00 AC per 39 lines of code.

Figure 3 brings two interesting information, (1) the prevalence of AC types across libraries (bottom) and (2) the distribution of AC types over all occurrences of ACs (top). On one hand, concerning (1), we found that *Conditional Operator* and *Logic as Control Flow* types had 100% of prevalence. The *Pre-Increment/Decrement* and *Infix Operator Precedence* types reached 81,48% and 70,37% of prevalence, respectively. *Type Conversion* and *Post-Increment/Decrement* types achieved average prevalence rate, with 66,67% and 59,26%, respectively. The *Arithmetic as Logic* and *Repurposed Variables* types reached both 11,11% of prevalence, the lowest rate. On the other hand, regarding (2), the *Logic as Control Flow*, *Infix Operator Precedence*, and *Conditional Operator* types represent together more than 86% of all occurrences. In contrast, *Pre-Increment/Decrement*, *Change of Literal Encoding*, *Repurposed Variables*, and *Arithmetic as Logic* types combined represent less than 2,50% of all ACs occurrences.

Figure 4 shows the absolute number of occurrences of each AC type per library. This figure cross the information we first presented separately in Figure 3. There it is possible to visualize how the ACs types are diffused across libraries and the number of each AC type occurrence in every studied library. Thus, it is not difficult to see that *Logic as Control Flow*, *Conditional Operator*, and *Infix Operator Precedence* shows high prevalence in both cases, presence across libraries and overall number of occurrence, while *Arithmetic as Logic* and *Repurposed Variables* achieved the lowest rate in both.

Figure 5 presents the proportion of classes with and without ACs for each library. We observed that 23 of the 27 libraries had ACs in more than 20% of their classes. Functor has the highest number of classes containing ACs (111 out 158). Over 50% of the classes in Functor library had at least one AC. On the other hand, Hamcrest had the lowest number of classes with ACs (9 out 80), only 11.2% of the total classes.

We found a strong correlation[7] between the number of LoC and the number of ACs ($r = 0.9244$) and a high degree of correlation between the number of classes and the number of ACs ($r = 0.7793$). For instance, the Math library has the largest number of classes containing ACs (390 of 990 classes). Also, it has the largest absolute number of occurrences of ACs (4.174), as well as the largest number of lines of code (100.364). It also contains the largest number of ACs types

[7]We computed only Pearson's correlation in this study.

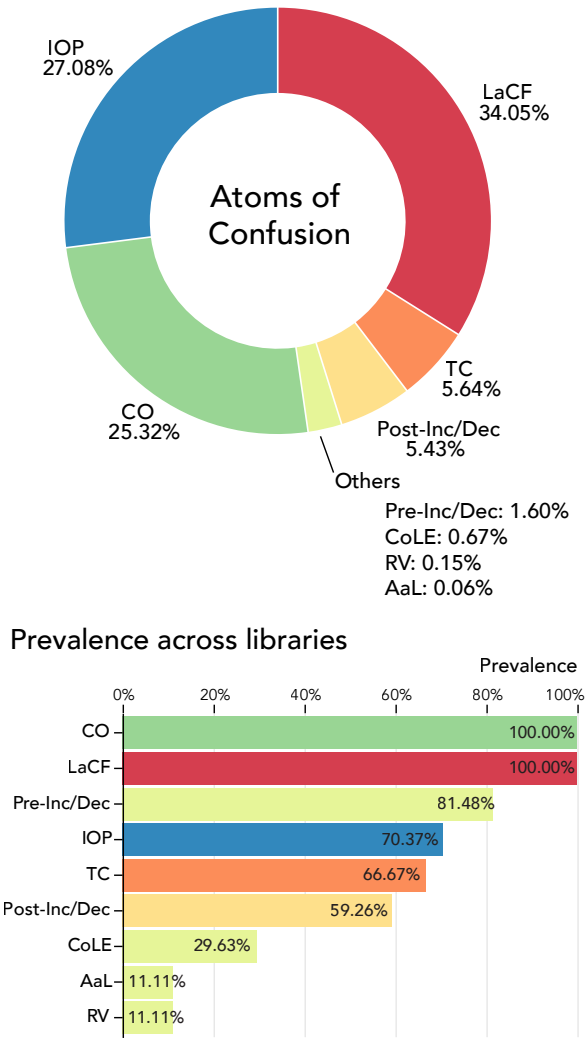

Fig. 3. Summary of ACs' prevalence. Top: Distribution of AC types over all the occurrences of ACs. Bottom: Prevalence of AC types across the studied libraries.

found (9), tied with the Compress library, the second-largest library in terms of lines of code (44.730). The Proxy library possesses the fewest number of ACs occurrences (15), and is the second smallest library in terms of lines of code (2.072). Finally, the DBCP and DbUtils libraries have the fewest number of distinct ACs types (2).

Table IV also shows each AC type's occurrences found in the libraries. The most common type was the *Logic as Control Flow* with 3.800 occurrences. This AC was also quite frequent in the C and C++ projects analyzed by Gopstein et al. in [10]. The wide usage of *Logic as Control Flow* AC may be due to the short form of expressing a conditional structure, rather than using the `if-else` statement [14].

Infix Operator Precedence was the second most common AC, with 3.148 occurrences. The Math library was the main responsible for this result. Miscellaneous math-related methods in this library contributed to a multiplicity of occurrences for this AC. There were 2,753 occurrences in the Math project alone, while the second library with the most occurrences of this AC, the Lang library, had only 138 occurrences. This AC was also the second most common in the C and C++ projects studied in [10]. *Infix Operator Precedence* is encouraged to some extent by the software engineering community. It is common for IDEs and code formatters to offer a feature for removing "unnecessary" parentheses. Unfortunately, this ends up automatically adding ACs in the source code [10].

The *Conditional Operator* AC was the third most frequent atom, with 2.874 occurrences, as also observed in [10]. In that study, it was also one of the most common ACs. The *Conditional Operator* AC is also encouraged by the software engineering community. Kernighan and Pike in [13] state that the use of the ternary operator (`<condition> ? <expression> : <expression>`) is good for short expressions. In a similar prevalence study, the authors omitted the *Conditional Operator* AC in their experiment because of its high number of occurrences in practice [18].

We did not find the *Omitted Curly Braces* AC in any studied library, in contrast to what was observed in [10]. In that study, the *Omitted Curly Braces* was the most common AC in C and C++ projects. However, omitting curly braces is not always considered a bad practice in general. One of the projects analyzed by Gopstein et al. [10] was the Linux operating system and the Linux kernel coding style recommends that regarding placing braces: "*Do not unnecessarily use braces where a single statement will do*" [12]. In Java, on the other hand, conventionally, the use of curly braces is encouraged in the code standards defined by Apache and Google. The Apache Commons Coding Standards states that: "*Brackets should begin and end on a new line and should exist even for one-line statements*" [8]. The Google Java Style Guide states: "*Braces are used with* `if`, `else`, `for`, `do` *and* `while` *statements, even when the body is empty or contains only a single statement*" [17]. Thus, the different development contexts of the Java and C/C++ languages may explain the big difference in the results of this AC prevalence.

The *Repurposed Variables* AC had a significant frequency in [10] in contrast to what we observed in our study, in which this AC was rare, with just 18 occurrences. The *Arithmetic as Logic* AC was also rare in our study with only 7 occurrences but, this AC was not included in the study of Gopstein et al..

B. RQ2. To what extent do different types of atoms of confusion co-occur, at the class level, in long-lived Java libraries?

We measured the co-occurrence of atoms in the same class. Our goal was to observe tendencies for certain ACs to occur together in the same Java file. Figure 7 presents a co-occurrence matrix of atoms of confusion at the class level for all libraries. We learned that the *Conditional Operator, Logic as Control Flow* and *Infix Operator Precedence* ACs are more likely to co-occur in the same class. These results confirm a trend pointed out by the RQ1 results, as these three atoms that co-occur more frequently at the class level are also the three most common atoms in the libraries studied.

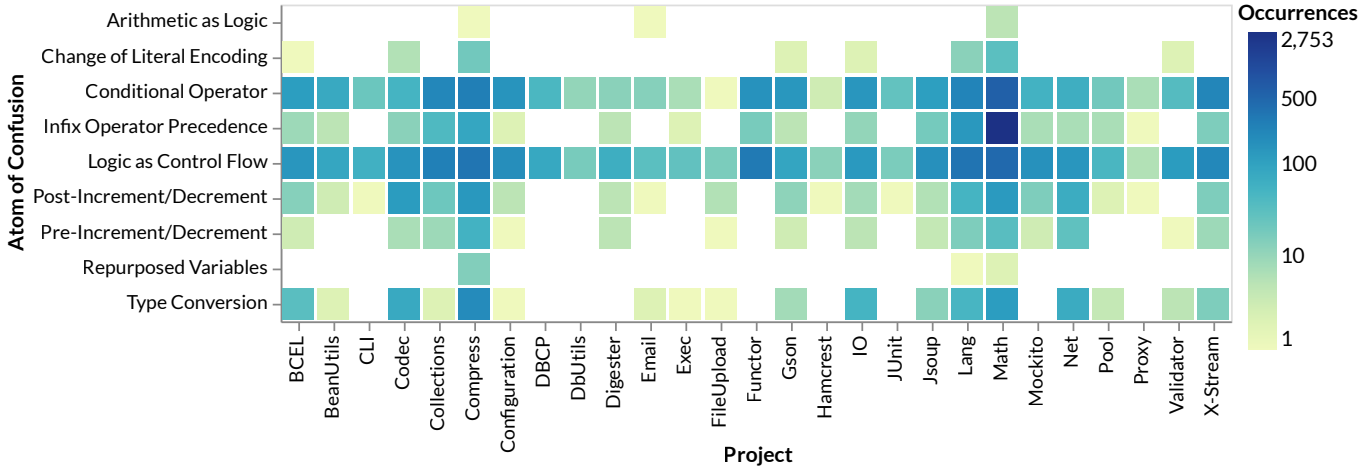

Fig. 4. The absolute number of occurrences of ACs per library.

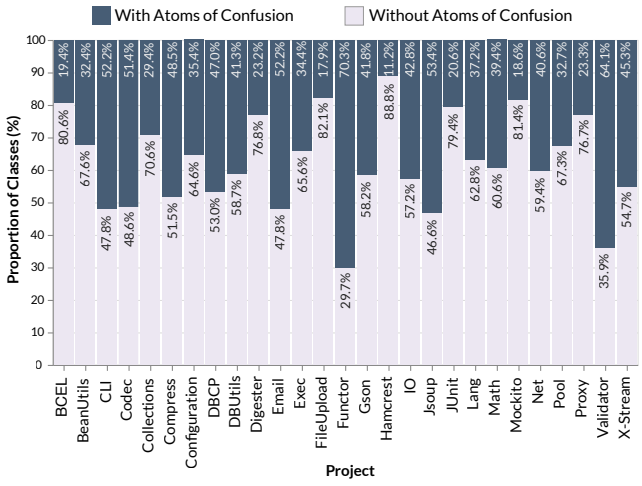

Fig. 5. Proportion of classes with and without ACs per library.

Once again, we can see the influence of the Math library on these results, boosting the *Infix Operator Precedence* numbers in this analysis. De facto, we found a strong correlation between the number of LoC and the number of AC co-occurrences ($r = 0.9130$) and a high degree of correlation between the number of classes and the number of ACs ($r = 0.7538$). However, we found a low degree of correlation between the number of contributors in the repository and the number of AC co-occurrences ($r = 0.0623$).

The *Arithmetic as Logic* AC had the lowest numbers of co-occurrences. We expected this behavior since, as also shown by the RQ1 results, this atom was the least common in the libraries. The *Arithmetic as Logic* AC co-occurred only with the *Conditional Operator* and *Infix Operator Precedence* ACs.

We found 7 different ACs in a single class in the Compress and Math libraries. Moreover, Lang, Net, and Jsoup libraries had 6 AC types. Not so differently, we observed classes with 5 AC types in the Codec, Collections, and Gson projects.

Code Snippet 1
Source: Mockito
Atoms of Confusion
2 Logic as Control Flow

```
return invocation.getMock() == candidate.getMock()
        && hasSameMethod(candidate)
        && argumentsMatch(candidate);
```

Code Snippet 2
Source: Collections
Atoms of Confusion
2 Conditional Operator

```
return (v1 ^ v2) ? ( (v1 ^ trueFirst) ? 1 : -1 ) : 0;
```

Code Snippet 3
Source: Compress
Atoms of Confusion
3 Infix Operator Precedence

```
channel.position(entry.localDataStart + 3
  * WORD + 2 * SHORT + nameLen + 2 * SHORT);
```

Code Snippet 4
Source: Functor
Atoms of Confusion
1 Logic as Control Flow
1 Conditional Operator

```
return (null == getCondition()
            ? null == that.getCondition()
            : getCondition().equals(that.getCondition()))
            && (null == getAction()
                    ? null == that.getAction()
                    : getAction().equals(that.getAction()));
```

Code Snippet 5
Source: Math
Atoms of Confusion
1 Infix Operator Precedence
1 Conditional Operator

```
final T[] aDotI = (2 * i - 1) < a.length ? a[2 * i - 1] : null;
```

Fig. 6. Atoms of confusion co-occurrence code snippets.

Figure 6 shows code snippets extracted from the studied libraries in which our tool detected a co-occurrence of ACs. The code snippet 1 was extracted from the Mockito library and presented two *Logic as Control Flow* ACs. The code snippet 2, from the Collections library, shows two *Conditional Operator* ACs. The code snippet 3 has three *Infix Operator Precedence* ACs in the Compress library. The code snippet 4, which our tool found in the Functor project, contains one *Logic as Control Flow* AC and one *Conditional Operator* AC. Finally, in the code snippet 5, extracted from Math, we have

one *Infix Operator Precedence* and *Conditional Operator* ACs.

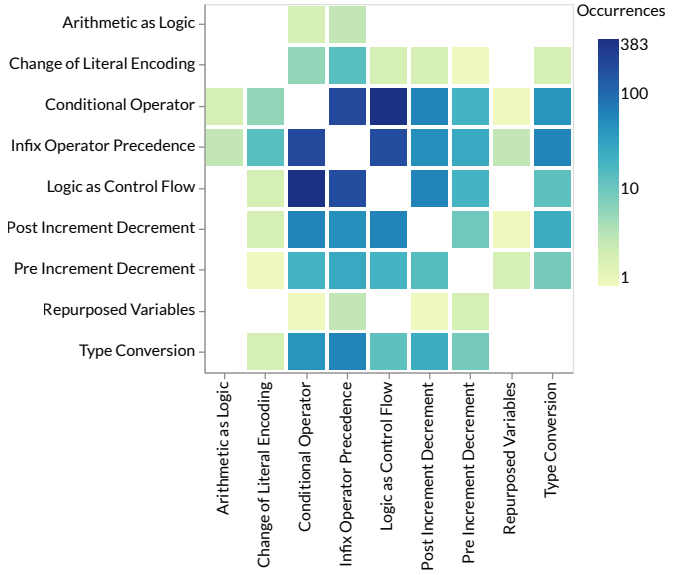

Fig. 7. Atoms of confusion co-occurrence matrix for all libraries.

C. RQ3. How long do atoms of confusion survive in long-lived Java libraries?

Finally, we studied the prevalence evolution of the ACs in 24 libraries. As mentioned before, we did not evaluate three libraries (i.e., Functor, Proxy, and Hamcrest) in this phase. This is because we didn't find enough versions of these three libraries to assess the ACs' evolution over time. Table V shows data from the first and last versions of the 24 libraries. The analysis covered a total of 455 releases. While Gson and Jsoup libraries had the highest number of versions analyzed (39 and 38), Exec and CLI libraries had the lowest version numbers (5 and 7).

Our tool detected the presence of ACs since the first releases of the 24 libraries. We observed that as libraries have grown in size (LoC), the presence of ACs also has increased. Only the JUnit library decreased the number of ACs since the project had a 41.55% reduction in size (LoC).

The Math and Compress libraries had the highest insertion of ACs between the first and last versions analyzed. Math, for example, currently has 4009 more ACs than the first release analyzed. On the other hand, the Exec, DBUtils, and FileUpload libraries had the smallest absolute increase. DBUtils, for instance, has 13 more ACs, although developers had added more than two thousand LoC.

In 12 libraries, the number of ACs grew proportionately more than the growth of their LoCs. For example, the Gson library had a 1,778.57% increase in the presence of ACs and its LoC number only increased by 222.96%. However, in 10 libraries, this growth was relatively lower. For example, DBUtils library increased its number of classes by 170.59%, but the number of ACs only augmented by 81.25%.

Table VI shows the evolution of the ratio number of ACs to the number of LoCs over time. Also, we observed the spread of ACs in library classes over time. The libraries indicated different behaviors for the two variables observed. In 11 libraries, the ratio of ACs to LoCs decreased. In the case of the Codec and Validator libraries, this reduction was more significant than 50%. The curves of these projects indicate a decrease, almost constant, of this ratio over time. On the contrary, 13 libraries had a growth in this ratio. Four of them, BeanUtils, CLI, Gson, and X-Stream, showed an increase of ACs to LoCs greater than 100%.

We observed a particular behavior in 8 libraries regarding the percentage of classes with ACs. The variation of the ratio of classes with ACs between the first and last versions was inferior to 7%. Although the phenomenon seems stable comparing just the first and later versions, there was variation over time in these eight libraries. For example, the CLI, Compress, Exec, and Jsoup projects had both increases and decreases in this variable over time.

Eleven libraries increased the percentage of Java classes with ACs in the observed period. Four libraries had more than 100% increases: BeanUtils, IO, Gson, and X-Stream. Moreover, the percentage grew practically in the last two libraries with each new version. On the other hand, in five libraries, we saw a reduction of more than 10%. The most notable case was of the Collection library, in which in the first analyzed version, there were 53% of classes with ACs and, in the last version, 29,54%.

VI. Further Discussion and Implications

A. Results Discussion

As we mentioned, Gopstein et al. [9] introduced the concept of **Atom of Confusion** (AC). Previous work has shown that ACs can affect code comprehension and hinder software maintenance and evolution in C and C++ projects. From the Java code patterns of these atoms [9, 15], our study found 11.404 occurrences in the 27 projects studied. Our results showed that 23 of the 27 analyzed systems had atoms in more than 20% of their classes (RQ1). There was a presence of atoms of confusion in all the analyzed projects. The Conditional Operator and Logic as Control Flow were present in all the libraries studied, while *Arithmetic as Logic* and *Repurposed Variables* appeared in only three projects. Moreover, our tool did not find *Omitted Curly Braces* occurrences.

Concerning the co-occurrence of ACs at the class level, we observed that there is a tendency for certain AC types to occur together in the same class (RQ2). For instance, the ACs *Conditional Operator*, *Logic as Control Flow* and *Infix Operator Precedence* are more likely to co-occur in the same class. This phenomenon may be related to the code style of developers who modified the same class.

Finally, in the analysis of ACs evolution over time (RQ3), we observed that the number of ACs increased. However, this phenomenon did not occur similarly in the analyzed projects. In ten projects, the number of ACs grew more than the size of the system. In other projects, there was a decrease in the ACs number per LOC. It is noteworthy that, in the way we studied prevalence evolution, we can only

TABLE V
EVOLUTION OF ACS AND LOCS IN THE 24 PROJECTS

	First Release				Last Release				Variation		
Library	**Version**	**Classes**	**LoCs**	**ACs**	**Version**	**Classes**	**LoCs**	**ACs**	**Classes**	**LoCs**	**ACs**
BCEL	5.2	335	23.631	276	6.5.0	391	31.686	322	16.72%	34.09%	16.67%▲
BeanUtils	1.5	62	5.196	34	1.9.4	111	11.644	174	79.03%	124.10%	411.76%▲
CLI	1.0	18	1.498	29	1.5.0	23	2.151	84	27.78%	43.59%	189.66%▲
Codec	1.1	14	937	107	1.15	72	9.313	436	414.29%	893.92%	307.48%▲
Collections	1.0	26	4.326	90	4.4	326	28.955	565	1153.85%	569.33%	527.78%▲
Compress	1.0	61	7.437	229	1.2.1	359	44.730	1.155	488.52%	501.45%	404.37%▲
Configuration	1.0	29	5.229	57	2.7	260	28.011	342	796.55%	435.69%	500%▲
DBCP	1.0	32	4.349	68	2.9.0	66	14.454	127	106.25%	232,35%	86,76%▲
DbUtils	1.0	17	1.002	16	1.7	46	3.074	29	170.59%	206,79%	81,25%▲
Digester	1.5	37	3.631	37	3.2	168	9.917	94	354.05%	173.12%	154.05%▲
Email	1.0	9	1.338	17	1.5	23	2.815	50	155.56%	110.39%	194.12%▲
Exec	1.0	29	1.675	33	1.3	32	1.757	38	10.34%	4.90%	15.15%▲
FileUpload	1.0	11	1.230	12	1.4	39	2.425	26	254.55%	97.15%	116.67%▲
IO	1.0	34	2.041	48	2.11.0	180	14.024	358	429.41%	587.11%	645.83%▲
Lang	1.0	26	4.319	100	3.12.0	215	29.745	880	726.92%	588.70%	780.00%▲
Math	1.0	106	7.162	165	3.6.1	990	100.364	4.174	833.96%	1301.34%	2429.70%▲
Net	1.0.0	103	8.714	132	3.8.0	212	20.199	389	105.83%	131.80%	194.70%▲
Pool	1.0	19	1.713	16	2.11.1	49	5.905	80	157.89%	244,72%	400%▲
Validator	1.0	17	1.874	87	1.7	64	7.619	167	276.47%	306.56%	91,95%▲
Gson	1.0	54	2.583	14	2.8.9	77	8.342	263	42.59%	222.96%	1,778.57%▲
Jsoup	0.1.1	25	2.079	78	1.14.3	73	13.714	323	192.00%	559.64%	314.10%▲
JUnit	4.12	195	9.317	104	5.8.2	95	5.446	45	-51.28%	-41,55%	−56,73%▼
Mockito	2.25.0	453	15.920	208	4.3.0	467	20.298	249	3.09%	27.50%	19.71%▲
X-Stream	0.2	50	1.235	12	1.4.19	361	21.859	502	622.00%	1669.96%	4083.33%▲

confirm that the number of ACs inserted over time was more significant than the number of ACs removed. Even so, it is interesting to note that its occurrence has not decreased (in absolute terms) in these systems. As already stated in previous work, ACs negatively impact code readability; their presence probably affects developers during maintenance tasks in these 27 libraries.

B. Implications for Researchers

The presence of atoms of confusion in long-lived Java libraries grows over time. This phenomenon needs further investigation into why developers insert ACs into the code. For example, what are the causes (developers' experience? developer's code style?) and consequences of this phenomenon (bugs? time of maintenance? code readability?).

Furthermore, some types of ACs were prevalent and showed an increasing trend in the number of occurrences. However, other types of ACs are rare. In this sense, these results may influence the efforts to create tools focused on detecting and refactoring more prevalent ACs.

C. Implications for Practitioners

Previous work has shown that confusing code impacts code comprehension and, hence, the development process. Rahman in [22] observed that when programmers are involved in high comprehension effort, they navigate and make edits at a significantly slower rate. Ebert et al. in [7] observed that code reviewers often do not understand the change being reviewed or its context. Also in the context of software development, developers tend to understand certain code structures more slowly than other ones, e.g., `for` loops take more time to be understood than sequences of `if` [1]. As well as some programming practices also affect the code readability [6].In the context of ACs Gopstein et al. [10] showed a strong relationship between ACs and bug fix commits and also pointed out that atoms tend to be more commented. Hence, it is important to disseminate this knowledge among developers, alerting them to the presence of these ACs in code.

In this sense, developers can use our detection tool in continuous integration and code review processes to be aware of the existence of ACs. Additionally, IDEs plugins could use our tool to perform static code analysis, checking for the presence of atoms in the source code at the time of writing, even before this code is compiled and executed.

VII. THREATS TO VALIDITY

The threats to the validity of our investigation are discussed using the four threats classification (conclusion, construct, internal, and external validity) presented by Wohlin et al. [24].

A. Conclusion Validity

Threats to the conclusion validity are concerned with issues that affect the ability to draw correct conclusions regarding the treatment and the outcome of an experiment. To avoid this threat, we use known metrics already used in previous studies on the prevalence of code patterns in software [10] [18] [4]. Thus, we use the count, frequency and proportion of ACs in the studied software as metrics.

B. Internal Validity

Threats to internal validity can affect the independent variable concerning causality without the researcher's knowledge. Thus, they threaten the conclusion about a possible causal relationship between treatment and outcome. In this paper, we

TABLE VI
EVOLUTION OF ACS/LOCS AND CLASSES WITH ACS IN THE 24 PROJECTS

Library	First Release		Last Release		Variation		Evolution	
	ACs/ LoCs	Classes with AC	ACs/ LoCs	Classes with AC	ACs/ LoCs	Classes with AC	ACs/LoCs	Classes with AC
BCEL	0.01168	18.51%	0.01016	19.44%	-12.99%	5.02%		
BeanUtils	0.00654	11.30%	0.01494	32.40%	128.36%	186.73%		
CLI	0.01936	50.00%	0.03905	52.20%	101.72%	4.40%		
Codec	0.11416	42.90%	0.04682	51.40%	-58.99%	19.81%		
Collections	0.0208	53.80%	0.01951	29.40%	-6.20%	-45.35%		
Compress	0.03079	47.50%	0.02582	48.50%	-16.14%	2.11%		
Configuration	0.0109	48.30%	0.01221	35.40%	12.01%	-26.71%		
DBCP	0.01563	28.10%	0.00879	47.00%	-43.80%	67.26%		
DbUtils	0.01597	35.30%	0.00943	41.30%	-40.92%	17.00%		
Digester	0.01019	27.00%	0.00948	23.20%	-6.98%	-14.07%		
Email	0.0127	55.60%	0.01776	52.20%	39.80%	-6.12%		
Exec	0.0197	34.50%	0.02163	34.40%	9.78%	-0.29%		
FileUpload	0.00976	27.30%	0.01072	17.90%	9.90%	-34.43%		
IO	0.02352	17.60%	0.02553	42.80%	8.55%	143.18%		
Lang	0.02315	50.00%	0.02959	37.20%	27.78%	-25.60%		
Math	0.02304	29.20%	0.04158	39.40%	80.50%	34.93%		
Net	0.01515	31.10%	0.01926	40.60%	27.13%	30.55%		
Pool	0.00934	21.10%	0.01355	32.70%	45.05%	54.98%		
Validator	0.04643	58.80%	0.02192	64.10%	-52.78%	9.01%		
Gson	0.00542	16.70%	0.03153	42.90%	481.65%	156.89%		
Jsoup	0.03752	52.00%	0.02355	53.40%	-37.24%	2.69%		
JUnit	0.01116	22.10%	0.00826	20.60%	-25.97%	-6.79%		
Mockito	0.01307	18.80%	0.01227	18.60%	-6.11%	-1.06%		
X-Stream	0.00972	20.00%	0.02297	45.40%	136.38%	127.00%		

do not seek to demonstrate casual relationships but only to discuss occurrences and co-occurrences of ACs. Hence, this kind of threat does not apply to our study.

C. Construct Validity

Construct validity concerns generalizing the result of the study to the concept or theory behind the study. We adopted a peer debriefing approach for research design validation and document review. Our goal was to avoid inconsistencies in the interpretation of the results. Additionally, we developed a tool that automates our study's data collection, seeking to prevent or alleviate the occurrence of human-made mistakes in this stage. To improve the confidence in our tool, we also evaluated its precision and recall looking to avoid bias caused by possible false positives and false-negatives results.

D. External Validity

Threats to external validity are conditions that limit our ability to generalize the results of our study to industrial practice. The main threats to this validity are related to the domain and sample size (i.e., the 27 open-source projects) we used in this study. Concerning the sample domain, we try to deal with this threat by arguing that those projects present several usage scenarios. Additionally, concerning the sample size, we dealt with this threat using diversity and longevity criteria. We chose Apache Commons and picked up other well-known libraries developed by different teams to get more diversity regarding team knowledge, skills, and coding practices. Finally, we chose open-source projects that are long-lived as a way to guarantee a degree of maturity and stability.

VIII. FINAL CONSIDERATIONS

In this study, we investigated the prevalence and evolution over the time of atoms of confusion in 27 open source long-lived Java libraries. In the prevalence analysis, our results showed that atoms of confusion were present in all the studied libraries. However, we also show a non-homogeneous presence of ACs in the projects. Three ACs were the most prevalent in almost all projects, and we rarely found some ACs. This work can aid developers to avoid writing source code that contains atoms, as it may lead to code comprehension-related problems during software maintenance and evolution.

In addition to the results of this work, we also provide essential infrastructure for conducting future research. We give a manfully verified dataset and a validated tool for identifying atoms of confusion in Java-based systems. This dataset enables the validation of new tools for atom identification, while our tool enables programmers to find and remove ACs from Java source code. All our materials are available in https://anonymous.4open.science/r/bohr-aoc-api-3E3D/.

In future work, we intend to study the impact of ACs on software quality attributes, such as bug occurrence, technical debit, code complexity, and maintainability effort.

REFERENCES

[1] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. Syntax, predicates, idioms—what really affects code complexity? *Empirical Software Engineering*, 24 (1):287–328, 2019.

[2] K.H. Bennett, V.T. Rajlich, and N. Wilde. Software evolution and the staged model of the software lifecycle. volume 56 of *Advances in Computers*, pages 1–54. Elsevier, 2002. doi: https://doi.org/10.1016/S0065-2458(02)80003-1. URL https://www.sciencedirect.com/science/article/pii/S0065245802800031.

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

[4] Francisco Gonçalves de Almeida Filho, Antônio Diogo Forte Martins, Tiago da Silva Vinuto, José Maria Monteiro, Ítalo Pereira de Sousa, Javam de Castro Machado, and Lincoln Souza Rocha. Prevalence of bad smells in pl/sql projects. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 116–121. IEEE Press, 2019. doi: 10.1109/ICPC.2019.00025. URL https://doi.org/10.1109/ICPC.2019.00025.

[5] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael

de Mello, Anderson Oliveira, Alessandro Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. Atoms of confusion: The eyes do not lie. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 243–252, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450387538. doi: 10.1145/3422392.3422437. URL https://doi.org/10.1145/3422392.3422437.

[6] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 277–285, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357142. doi: 10.1145/3196321.3196342. URL https://doi.org/10.1145/3196321.3196342.

[7] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion detection in code reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 549–553, 2017. doi: 10.1109/ICSME.2017.40.

[8] The Apache Software Foundation. Coding standards, 2022. URL https://commons.apache.org/proper/commons-net/code-standards.html.

[9] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 129–139, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106264. URL https://doi.org/10.1145/3106237.3106264.

[10] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of confusing code in software projects: Atoms of confusion in the wild. MSR '18, page 281–291, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196432. URL https://doi.org/10.1145/3196398.3196432.

[11] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion. ESEC/FSE 2020, page 605–616, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409714. URL https://doi.org/10.1145/3368089.3409714.

[12] The kernel development community. Linux kernel coding style, 2022. URL https://www.kernel.org/doc/html/v4.10/process/coding-style.html.

[13] Brian W Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley Professional, 1999.

[14] Chris Langhout. Investigating the perception and effects of misunderstandings in java code. Master's thesis, Delft University of Technology, 2020.

[15] Chris Langhout and Maurício Aniche. Atoms of confusion in java, 2021.

[16] Luan Lima, Lincoln Rocha, C. I. M. Bezerra, and Matheus Paixao. Assessing exception handling testing practices in open-source libraries. *Empirical Software Engineering*, 26, 09 2021. doi: 10.1007/s10664-021-09983-3.

[17] Google LLC. Google java style guide, 2022. URL https://google.github.io/styleguide/javaguide.html#s4.1-braces.

[18] Flávio Medeiros, Gabriel Lima, Guilherme Amaral, Sven Apel, Christian Kästner, Márcio Ribeiro, and Rohit Gheyi. An investigation of misunderstanding code patterns in c open-source software projects. *Empirical Softw. Engg.*, 24(4):1693–1726, August 2019. ISSN 1382-3256. doi: 10.1007/s10664-018-9666-x. URL https://doi.org/10.1007/s10664-018-9666-x.

[19] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, 2015. doi: 10.1109/ICPC.2015.12.

[20] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46: 1155–1179, 2015. doi: 10.1002/spe.2346. URL https://hal.archives-ouvertes.fr/hal-01078532/document.

[21] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. A large-scale study on the usage of java's concurrent programming constructs. *Journal of Systems and Software*, 106:59–81, 2015. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.04.064. URL https://www.sciencedirect.com/science/article/pii/S0164121215000849.

[22] Akond Rahman. Comprehension effort and programming activities: Related? or not related? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 66–69, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196470. URL https://doi.org/10.1145/3196398.3196470.

[23] Spencer Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

[24] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435.

[25] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10): 951–976, 2018. doi: 10.1109/TSE.2017.2734091.