Atoms of Confusion in the Android Open Source Project: A Prevalence Study

Davi Tabosa[1], **Windson Viana**[1], **Lincoln Rocha**[1]

[1]Universidade Federal do Ceará (UFC)
60.440-900 – Fortaleza – CE – Brazil

davitabosa12@alu.ufc.br, windson@virtual.ufc.br, lincoln@dc.ufc.br

Abstract. *The Android Open Source Project (AOSP) is the open-source project responsible for the Android operating system. In the AOSP, developers collaborate to add features, fix bugs, and improve the performance of a code base of more than 14 million lines written in different programming languages (e.g., C, Java, and Kotlin). This code must have minimum quality requirements to facilitate its maintenance and evolution. However, latent software quality problems can persist in such complex projects, leading to maintainability and evolution issues. An example of a latent problem is Atoms of Confusion (AC), small indivisible code fragments that cause comprehension difficulties. To shed light on this matter, we conducted an empirical study in the AOSP with twofold goals: (i) perform a prevalence and frequency analysis of ACs in this project, and (ii) relate ACs presence with Chidamber & Kemerer (CK) metrics suite. We found that 331 of the 370 AOSP repositories analyzed have at least one atom, with Logic as Control Flow being the most frequent and prevalent, with more than 110,000 occurrences found and present in 96% of the repositories with the presence of atoms. We also observed that the presence of ACs has a positive correlation with some CK metrics, such as WMC (Weighted Methods per Class) and RFC (Response for a Class), as well as the number of LOC (Lines of Code).*

1. Introduction

In open-source projects, writing software with a team of developers from many parts of the world can be challenging. They must agree on code standards, language conventions, and code styles while keeping it easier to read, understand, and maintain for future work. Some studies investigate how to make the code more maintainable and understandable. One of those studies is about Atoms of Confusion[1] (AC), considered by researchers as the smallest snippets of code that may introduce some misunderstanding for developers.

Gopstein *et al.* [5] started identifying those atoms of confusion by proposing a collection of snippet candidates and validating them in a study to evaluate if a programmer gets confused by looking at a code with ACs and without ACs. Prevalence studies are common while researching Atoms of Confusion. Another study, conducted by Gopstein *et al.* [6], investigated the prevalence of ACs in common C/C++ projects. Langhout and Aniche [7] extended this study to investigate the Java programming language, porting the original set of ACs into Java. Mendes *et al.* [8] used this ported set of ACs to investigate the prevalence over time on long-lived Java projects. They found out that the presence

[1]https://telesto.poly.edu/

of ACs increases over time when the project also increases in size. Pinheiro *et al.* [10] expanded Mendes *et al.* study to explore the addition and removal of ACs during software evolution. Feijó *et al.* [4] studied the impact of the adoption of Continuous Integration and Continuous Delivery (CI/CD) on the rate of ACs. They found that the CI/CD adoption does change the rate of ACs. However, they have not found any statistically relevant relationship before and after the adoption of CI/CD.

Beyond studying ACs as a source of code confusion, some studies have investigated the relationship between ACs and bugs. Bogachenkova *et al.* [2] studied the relationship between pull request comments and the confusion that should have been raised by ACs, yielding no statistical relationship between them. However, Gosptein *et al.* [6] found a strong correlation between bug-fix commits and the removal of ACs. In turn, Pinheiro *et al.* [10] identified 9 removal cases where ACs were the direct cause of bug-fix and improvement commits. Although there is no consensus in the literature, the relationship with bugs leverages the study of ACs even more important.

Our research aims to study ACs in the Android ecosystem. Android is the most used operating system for smart devices, used in over 3.5 billion smartphones alone. The Android development is led by Google, which in collaboration with many independent developers around the globe, contributes to the open-source version of the operating system, called Android Open Source Project (AOSP). The project is organized in many layers with different repositories for each layer, such as the Android Core APIs, vendor drivers, hardware abstraction layer, kernel services, core applications, and other third-party libraries. Inside those repositories are source code files responsible for the operating system behavior, documentation, and build files. Thus, developing and maintaining a big project like this, spanning many parts of the Android ecosystem, is a great challenge.

The goal of this paper is two-fold. First, to investigate the prevalence and frequency of atoms of confusion in AOSP, comparing the results found across its different layers. We expect that the distribution of ACs across layers may vary enough to be easily distinguishable in comparison with each other. Second, correlate the presence of ACs with the Chidamber & Kemerer (CK) metrics suite as a proxy for maintainability.

This paper is organized as follows: Section 2 presents the background and related work. The study design and methodology are detailed in Section 3. The results are presented in Section 4. Section 5 discusses our findings and introduces the threats to study validity. Finally, Section 6 concludes the paper.

2. Background and Related Work

2.1. Atoms of Confusion in Java

As previously discussed, Atoms of Confusion are small, indivisible snippets of code that may cause misunderstandings while working on a source code. In the original work by Gopstein *et al.* [5], they proposed 19 types of ACs for C/C++. For example, the Conditional Operator is an atom that is comprised of an expression that usually contains a question mark and colon symbols (`expression ? value-if-true : value-if-false`) to write a shorter version of an `if-then-else` statement. According to Gopstein *et al.* [5], the presence of a Conditional Operator may raise code reading misunderstanding. When translating the study to Java, Langhout and Aniche [7]

managed to port 14 out of 19 types of ACs due to being related to features that are not present in Java, like preprocessor macros and other language syntax choices.

Extending the work by Langhout and Aniche, Mendes *et al.* [8] created an automated tool based on Spoon [9] for finding ACs in Java files. Table 1 shows the list of ACs that are detectable using the tool, that can also identify the snippet of code that contains the atom. This tool was used in 27 Java libraries to analyze the prevalence and occurrence of ACs over time. This study claims that the most prevalent atom types are Logic as Control Flow and Conditional Operator, found in all libraries under analysis. Logic as Control Flow is also the most common AC found throughout the libraries with 3,800 occurrences, followed by Infix Operator Precedence with 3,148 instances.

Pinheiro *et al.* [10] used the previously mentioned tool to find ACs in 21 Java projects in the Apache Commons ecosystem. The focus of this study was to analyze how Atoms of Confusion are introduced and removed in the lifecycle of those projects, linking the presence of ACs to issue tracker and bug tracker information. This study found instances of fixed bugs in which the root cause was an introduction of an AC.

Table 1. Atoms of Confusion in Java adapted from [8]

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]);`

2.2. Android Ecosystem Architecture

The Android Open Source Project is divided into 6 layers[2]:

1. **Application layer:** The layer closest to the end user. In this layer, code is written primarily in Kotlin or Java, using the Android API to interact with the operating system.
2. **System services layer:** Layer that accommodates components that expose functionality from the hardware or the Hardware Abstraction Layer to the Android API.
3. **Android Runtime:** It is the runtime that converts Java bytecode instructions to hardware-dependent machine code.

[2] https://source.android.com/docs/core/architecture

4. **Hardware Abstraction Layer:** It is the hardware abstraction layer that hardware manufacturers that use Android must implement to make the operating system interact with the hardware.
5. **Native Daemons and Libraries** contains daemons and other libraries that interact directly with the kernel. The logging system and the init system are examples of solutions in this layer.
6. **Linux Kernel** Android uses the Linux kernel to communicate with the hardware in use. In the codebase, AOSP is divided into hardware-independent and vendor-specific code.

2.3. AOSP Atoms of Confusion Dataset

We created a dataset[3] of Atoms of Confusion in Java files in the AOSP [11]. The dataset includes the instances of each AC found for each Java file across all projects and some usual metrics used in object-oriented development, like the ones proposed by Chidamber & Kemerer [3]. We used a Spoon-based tool to detect the presence of ACs and the metrics were extracted using the CK tool [1]. Each AC and metric found in the dataset are related to a single source code file, which is linked to a repository or "project" and this repository belongs to one of the layers in the AOSP. Using the relations in the database, it is easy to obtain metrics for a single project or aggregate those results across the layers in the AOSP.

The snapshot obtained from the AOSP at the time it was analyzed has over 29 million lines of code written in Java, across over 125 thousand files spanning from 370 repositories with at least one Java source code file present, as shown in Table 2.

Table 2. Overall AOSP information adapted from [11].

Data	Value
Size on disk without binaries	250 GB
No. repositories	1,293
No. repositories with Java files	370
No. Java files	125,687
Java Files without ACs	86,704
Java Files with ACs	38,983
Lines of Java code	29,855,095

3. Methodology

3.1. Research Objectives

The primary goal of this research is to investigate the prevalence of ACs in the AOSP, comparing the results found with other similar studies regarding the prevalence of ACs in Java projects. The secondary goal is to compare the results of prevalence for a particular layer with another, e.g. comparing the prevalence of ACs in the *frameworks* layer with the *system applications* layer. We expect to reproduce the results found by Gopstein *et al.* [5] that claim that projects in the same domain have similar prevalence behavior.

[3]https://doi.org/10.5281/zenodo.10576122

3.2. Research Questions

RQ1. *How are ACs distributed in the Android ecosystem?*

The purpose of this question is to know more about the presence of ACs in the AOSP, since this is a common data provided by many studies researching ACs in open-source projects. This research question also aims to obtain general statistics regarding AC behavior, like how many atoms were detected, and which layer has the most atoms.

RQ2. *How does the frequency and prevalence of ACs differ in AOSP compared to other Java systems?*

The goal of this research question is to compare our findings with other frequency and prevalence studies in Java projects, highlighting similarities and differences between them, the prevalence being if a particular AC is present in a project at least once and frequency is how many instances of this AC is present in a project.

RQ3. *How does the frequency and prevalence of ACs differ across AOSP layers?*

The purpose of this question is to replicate the findings of Gopstein *et al.* [5] It is crucial to learn how different are the code bases from the different layers of the AOSP. We expect that projects inside the same layer are more like each other in comparison with other projects from other layers.

RQ4. *What is the correlation between software quality metrics and the atoms of confusion in the ecosystem?*

We also aim to find the impact that atoms of confusion have in Java projects, observing commonly used metrics for object-oriented software that Chidamber & Kemerer compiled. Claiming that some AC raises or lowers some specific CK metric is out of the scope of this paper.

3.3. Study Design

We started our study by looking into the Android Atoms of Confusion Dataset and looking into the data present in each table. Inside the `ac_reports` table, we have found out that each row represents one AC for a specific file in a specific repository. In the projects table, we have noticed that the repositories were listed alongside their "layer". Knowing this, we have decided to study each project separately, calculating the prevalence and frequency of ACs. To obtain these values, we have performed an aggregation depending on its meaning. For example, the number of ACs across different projects would be aggregated by sum. Since most of the values present are simple counts, we decided to aggregate these by summing them together. To do these operations, we used Python scripts with the help of the pandas library.

After performing the aggregations, we started looking at the values of each layer and compared them against each other.

4. Results

4.1. RQ1 How are ACs distributed in the Android ecosystem?

Summary of RQ1: Almost every project has at least one atom of confusion. The most prevalent is Logic as Control Flow, present in 96% of the projects analyzed with atoms present.

Of the 370 projects analyzed, 331 had at least one AC present in the source code, totaling 89% of projects. Figure 1 shows the prevalence of atoms of confusion in the Android Open Source Project. One can see that the most prevalent atom type is Logic as Control Flow, which appears in 96% of the projects. Second is Conditional Operator, present in 91% of the projects, and third is Type Conversion, present in 71% of projects.

Observing the number of atoms found, we have discovered 327,226 instances of ACs. Logic as Control Flow is the most frequent atom occurring 110,593 times. We can assume that 1 in every 3 atoms found is Logic as Control Flow. The second most frequent is Type Conversion with 87,580 occurrences. The third most frequent type is Conditional Operator with 72,552 occurrences.

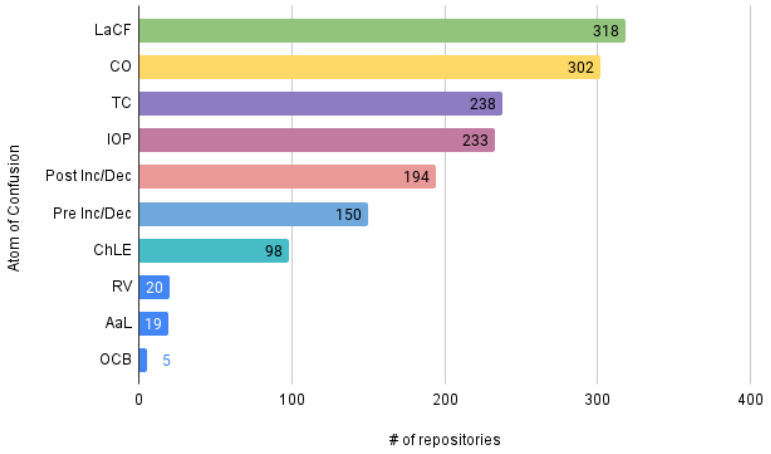

Figure 1. Prevalence of ACs across AOSP modules

4.2. RQ2 How does the frequency and prevalence of ACs differ in AOSP compared to other Java systems?

Summary of RQ2: Logic as Control Flow and Conditional Operator have equivalent results comparing AOSP and other projects. In the AOSP, however, the frequency of Type Conversion is like the frequency of Infix Operator Precedence in other projects.

To properly compare two big Java-based projects, we decided to investigate Mendes *et al.* study [8] as a comparison benchmark. In this study, the authors used the same AC detection tool as ours to investigate 27 long-lived Java projects, 21 of them belonging to the Apache Foundation.

Table 3 compares our findings with the results by Mendes *et al.* Logic as Control Flow was the most frequent AC in both the AOSP and in [8]. Conditional Operator and Post Inc/Dec atoms have also similar frequency values. However, Infix Operator

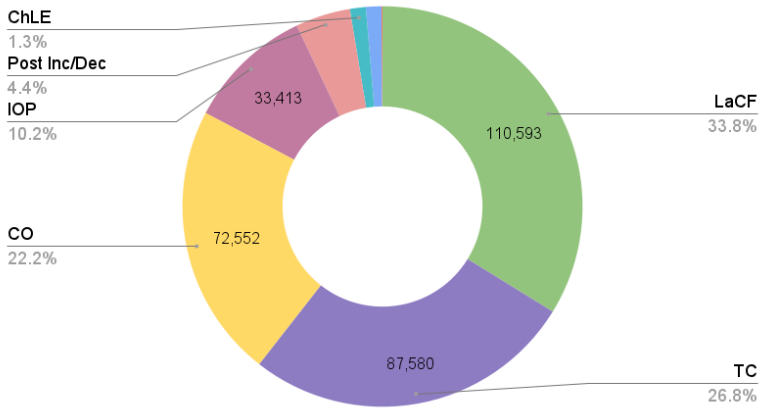

Figure 2. Frequency of ACs in the AOSP

Precedence appeared less than the other selection of Java projects and Type Conversion was more frequent. Table 4 compares the prevalence between the two studies and although they have different values, LaCF, IOP, CO, and TC managed to be the most prevalent atoms.

Study	LaCF	IOP	CO	TC	Post-Inc/Dec
AOSP	33.08%	10.2%	22.2%	26.8%	4.4%
Mendes *et al.*	34.05%	27.08%	25,32%	5.64%	5.43%

Table 3. Comparison of frequencies between AOSP and Mendes *et al.* list of projects

Study	LaCF	IOP	CO	TC	Post-Inc/Dec
AOSP	85.95%	62.97%	81.62%	64.32%	52.43%
Mendes *et al.*	100%	70.37%	100%	66.67%	59.26%

Table 4. Comparison of prevalence between AOSP and Mendes *et al.* list of projects

4.3. RQ3 How does the frequency and prevalence of ACs differ across AOSP layers?

Summary of RQ3: ACs were concentrated in the *packages*, *external*, and *frameworks* repositories, however, the distribution seems mostly similar overall.

Looking into Figure 3, it is noticeable that there is a concentration of ACs in *packages*, *external*, and *frameworks*. We can also observe that there are few differences in how they are distributed if we adjust the atom count by their lines of code.

4.4. RQ4 What is the correlation between software quality metrics and the atoms of confusion in the ecosystem?

We computed the Pearson correlation between the quantity of ACs and CK metrics (see Table 5). We noticed a slight positive correlation between Weighted Method Call, Number of Static Invocations, and Response for a Class. We also noticed a strong correlation between lines of code and number of ACs.

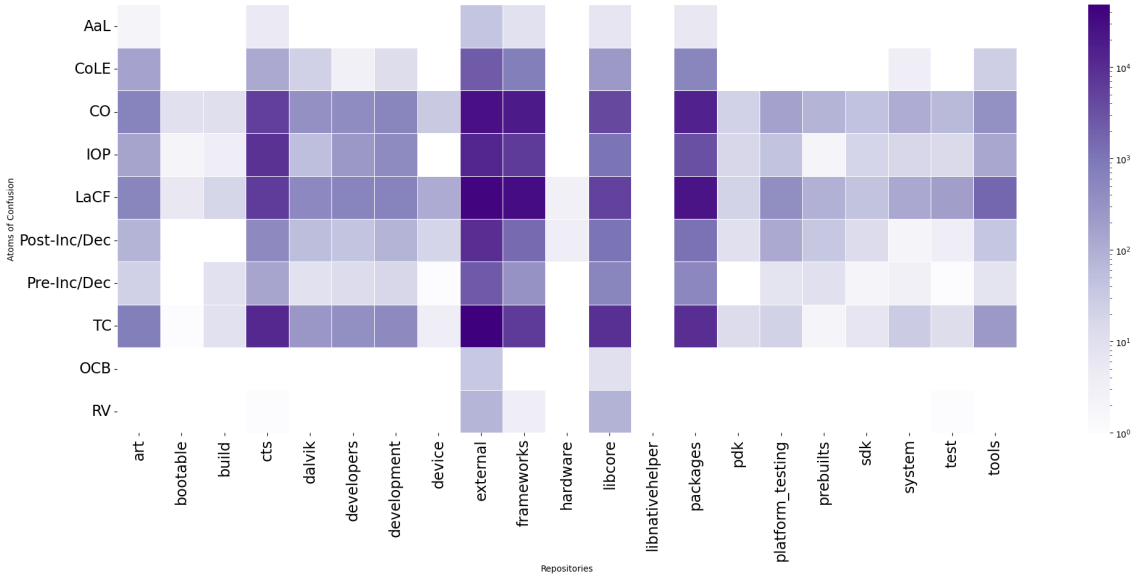

Figure 3. AC count for every root repository

Metric	Correlation
Lines of Code	0.94
Weighted Method Call	0.53
Number of Static Invocations	0.54
Response for a Class	0.50

Table 5. Correlation between AC count and Object-Oriented metrics

5. Discussion

5.1. Comparison to other AC studies

The results we found regarding prevalence and frequency were too close from the results found by [8], the difference being a higher presence of the Type Conversion atom. There are some files in AOSP that makes heavy use of the Type Conversion atom. For example, in the *external/conscrypt* project, casting to *byte* is highly frequent in some test files, as we can see on Figure 4. One can observe the outstanding strong relationship between the number of ACs and the number of lines of code, result that is also present in Gopstein *et al.* and Pinheiro *et al.* works [6, 10]. AOSP also has a lower frequency of the Infix Operator Precedence atom. Being a Google project, AOSP must follow the Google Java Style Guide that recommends, but not requires, putting optional parenthesis when using operators [4]. We can also observe in Figure 3 the lack of the Infix Operator Precedence in both *device* and *prebuilts* repositories.

5.2. Limitations and Threats to validity

5.2.1. Limitations

Our study uses a unique data source for analysis. Also, it contains a single version of the Android Open Source Project. So, it is not possible to draw conclusions regarding the evolution of the source code that may have taken place in the meantime.

[4]https://google.github.io/styleguide/javaguide.html#s4.7-grouping-parentheses

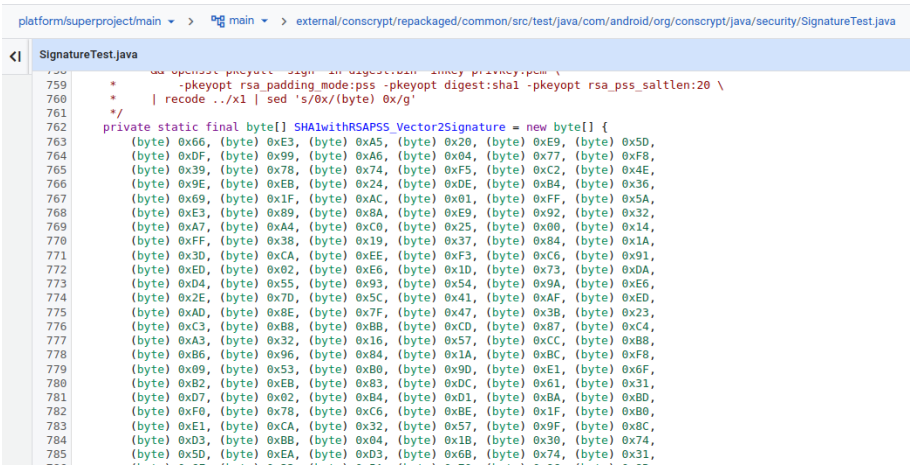

Figure 4. One of the test cases for Conscrypt

5.2.2. Threats to validity

Conclusion validity: A threat to Conclusion Validity is one that impacts the process of drawing sound conclusions. Our findings for RQ4 were based on performing a correlation test between the number of atoms found in a particular file against the measurements of the CK metrics. The values obtained are below the accepted threshold for a strong correlation ($>$0.6) for this paper. An argument can be made for considering the correlation to be strong, however after peer debriefing, we decided to take a more conservative approach.

Internal validity: A threat to Internal Validity is one that impacts the relationship between cause and effect in a study. Since the goal of this study is purely exploratory and does not aim to establish any relationship between ACs and object-oriented metrics, we believe that this threat is not applicable.

External validity: A threat to External Validity is one that impacts the generalization of the results obtained to future works in the field. Since the scope of this paper is focused on analyzing the AOSP exclusively, we believe that this threat is not applicable for this study.

6. Final Considerations

In this paper, we presented a comprehensive study of Atoms of Confusion in the Android Open Source Project. We have analyzed the prevalence and frequency of ACs, following tried and proven methodologies present in the state of art of AC studies, comparing our results with other studies and finding comparable results. We also made a correlation between the quantity of atoms and object-oriented software metrics, finding a moderate correlation between them and a strong correlation between number of atoms and number of lines of code.

Acknowledge

This research was partially funded by CNPQ, under grant 314425/2021-7

References

[1] M. Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[2] V. Bogachenkova, L. Nguyen, F. Ebert, A. Serebrenik, and F. Castor. Evaluating atoms of confusion in the context of code reviews. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 404–408, 2022.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[4] D. Feijó, C. de Almeida, and L. Rocha. Studying the impact of continuous delivery adoption on atoms of confusion rate in open-source projects. In *Anais do XI Workshop de Visualização, Evolução e Manutenção de Software*, pages 6–10, Porto Alegre, RS, Brasil, 2023. SBC.

[5] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, and J. Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 129–139, 2017.

[6] D. Gopstein, H. H. Zhou, P. Frankl, and J. Cappos. Prevalence of confusing code in software projects: Atoms of confusion in the wild. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 281–291, 2018.

[7] C. Langhout and M. Aniche. Atoms of confusion in java. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 25–35. IEEE, 2021.

[8] W. Mendes, O. Pinheiro, E. Santos, L. Rocha, and W. Viana. Dazed and confused: Studying the prevalence of atoms of confusion in long-lived java libraries. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 106–116. IEEE, 2022.

[9] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[10] O. Pinheiro, L. Rocha, and W. Viana. How they relate and leave: Understanding atoms of confusion in open-source java projects. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 119–130. IEEE, 2023.

[11] D. Tabosa, O. Pinheiro, L. Rocha, and W. Viana. A dataset of atoms of confusion in the android open source project. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 520–524. IEEE, 2024.