FEDERAL UNIVERSITY OF CEARÁ

CENTER OF SCIENCE

DEPARTAMENT OF COMPUTER SCIENCE

PROGRAM OF MASTER AND DOCTORATE IN COMPUTER SCIENCE

ACADEMIC MASTER'S DEGREE IN COMPUTER SCIENCE

WENDELL MILITÃO FERNANDES MENDES

STUDYING THE PREVALENCE OF ATOMS OF CONFUSION IN LONG-LIVED JAVA LIBRARIES

FORTALEZA

2022

WENDELL MILITÃO FERNANDES MENDES

STUDYING THE PREVALENCE OF ATOMS OF CONFUSION IN LONG-LIVED JAVA LIBRARIES

Dissertation submitted to the Program of Master and Doctorate in Computer Science of the Center of Science of the Federal University of Ceará, as a partial requirement for obtaining the title of Master Degree in Computer Science. Concentration Area: Software Engineering

Supervisor: Prof. Dr. Windson Viana de Carvalho

Co-supervisor: Prof. Dr. Lincoln Souza Rocha

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M492s Mendes, Wendell Militão Fernandes.
Studying the Prevalence of Atoms of Confusion in Long-Lived Java Libraries / Wendell Militão Fernandes Mendes. – 2022.
69 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2022.
Orientação: Prof. Dr. Windson Viana de Carvalho.
Coorientação: Prof. Dr. Lincoln Souza Rocha.

1. Empirical Study. 2. Program Comprehension. 3. Atoms of Confusion. 4. Long-lived Java projects. I. Título.

CDD 005

WENDELL MILITÃO FERNANDES MENDES

STUDYING THE PREVALENCE OF ATOMS OF CONFUSION IN LONG-LIVED JAVA LIBRARIES

Dissertation submitted to the Program of Master and Doctorate in Computer Science of the Center of Science of the Federal University of Ceará, as a partial requirement for obtaining the title of Master Degree in Computer Science. Concentration Area: Software Engineering

Approved on: 17/10/2022

EXAMINATION BOARD

Prof. Dr. Windson Viana de Carvalho (Supervisor)
Federal University of Ceará (UFC)

Prof. Dr. Lincoln Souza Rocha (Co-Supervisor)
Federal University of Ceará (UFC)

Prof. Dr. João Bosco Ferreira Filho
Federal University of Ceará (UFC)

Prof. Dr. Márcio de Medeiros Ribeiro
Federal University of Alagoas (UFAL)

I dedicate this work to my parents, Cilene and José Afonso, that have spared no effort to provide me with all the education I have.

ACKNOWLEDGEMENTS

As a child I never thought that one day I would achieve some of the things I have achieved throughout my life, among these things is this master's degree. Along my journey I realized how lucky I was to have met people along the way who helped and believed in me.

Firstly, I want to thank my parents, Cilene and José Afonso, immensely for the unconditional love and support I have always received, without this support I certainly wouldn't have gotten this far. I would like to thank them, at this moment, especially for all the sacrifices made to allow me to have the best education possible. I hope they are proud of me and of this new achievement.

A special thanks to my girlfriend, Raquel, for all the support and help she has given me throughout our relationship and also along the course of this master's program. Her companionship gave me strength in the most difficult moments. Being by her side makes me want to be better and go further.

Thanks to my supervisors Windson Viana and Lincoln Rocha for believing in me and for all the support and learning, thanks to Emanuele Santos and to my master's colleague student Oton Pinheiro for the help in the realization of this work and, also, to my colleagues from Master and Doctorate in Computer Science (MDCC), from Federal University of Ceará (UFC), who were always very helpful and supportive.

I would also like to thank my co-workers from Information Technology Superintendence (STI), from Federal University of Ceará (UFC), for supporting me and allowing me to conciliate my job with this research.

Finally, I thank everyone who made this work possible. You are present in every page of it, not only in these acknowledgements.

“Ninguém nasce feito. É experimentando-nos no mundo que nós nos fazemos.”

(Paulo Freire)

RESUMO

A atividade de compreensão do código-fonte é fundamental na manutenção e evolução de software, impactando em várias tarefas como: a correção de bugs, a reutilização de código e a implementação de novas funcionalidades. Um Átomo da Confusão (AC) é considerado a menor porção de código capaz de causar confusão em programadores, dificultando a correta compreensão de um código-fonte. Estudos anteriores mostraram que esses átomos podem ter um impacto significativo na presença de bugs em programas em C/C++ e aumentar o tempo e o esforço para a compreensão do código em sistemas C/C++ e Java. Para obter mais evidências sobre a difusão de ACs no ecossistema Java, essa pesquisa de mestrado realizou um estudo para analisar a prevalência, a co-ocorrência (a nível de classe), e a evolução de ACs em 27 bibliotecas tradicionais em Java. Para apoiar a investigação, foi desenvolvida uma ferramenta de pesquisa automática de ACs chamada BOHR. Esta ferramenta visa: (i) ajudar na identificação de ACs em sistemas Java; (ii) fornecer relatórios de prevalência desses ACs; e (iii) fornecer um API para o desenvolvimento de novos localizadores personalizados para a captura de novos ACs, bem como melhorar as identificações de átomos já implementadas. A ferramenta BOHR é capaz de detectar 10 dos 14 tipos de ACs apontados por Langhout e Aniche (LANGHOUT; ANICHE, 2021). Além da ferramenta, foi fornecido um conjunto de dados de projetos Java, anotado manualmente, utilizado para validar a precisão da ferramenta. Usando a ferramenta BOHR, foram encontradas 11.404 ocorrências nas bibliotecas estudadas. O *Conditional Operator* e o *Logic as Control Flow* foram os átomos mais prevalentes entre os 10 tipos de ACs avaliados. Observou-se que o *Conditional Operator* e o *Logic as Control Flow* foram mais suscetíveis a co-ocorrer em uma mesma classe. Por fim, a prevalência de ACs não diminuiu ao longo do tempo nos projetos analisados. Pelo contrário, em 13 bibliotecas, a presença cresceu proporcionalmente mais do que o tamanho da biblioteca em termos de linhas de código. Além disso, em 15 bibliotecas, a fração de classes Java contendo pelo menos um átomo também aumenta ao longo do tempo.

Palavras-chave: estudos empíricos; compreensão de código; átomos de confusão; projetos Java de longa duração.

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/C++ programs and increase the time and effort to code understanding in C/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 called BOHR. This tool aims to: (i) aid in the identification of ACs in Java systems; (ii) provide prevalence reports of these ACs; and (iii) provide an API for the development of new custom finders to capture new ACs, as well as improve already implemented ACs identifications. BOHR is able to detect 10 of the 14 types of ACs pointed out by Langhout and Aniche (LANGHOUT; ANICHE, 2021). We also provide a dataset, manually annotated, used to validate BOHR accuracy. Using BOHR, we 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.

Keywords: empirical study; program comprehension; atoms of confusion; long-lived Java projects.

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODES

LIST OF ABBREVIATIONS AND ACRONYMS

AaL	Arithmetic as Logic
AC	Atom of Confusion
ACs	Atoms of Confusion
CO	Conditional Operator
CoLE	Change of Literal Encoding
CSV	Comma Separated Values
IOP	Infix Operator Precedence
LaCF	Logic as Control Flow
LoC	Lines of Code
OCB	Omitted Curly Braces
Post-Inc/Dec	Post-Increment/Decrement
Pre-Inc/Dec	Pre-Increment/Decrement
RV	Repurposed Variables
TC	Type Conversion

SUMMARY

1 INTRODUCTION

1.1 Context and Motivation

Code comprehension is the activity in which software engineers seek to understand a computer program having its source code as the main reference (BENNETT *et al.*, 2002). 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 (RUGABER, 1995).

In software development, developers frequently deal with code snippets they had not written themselves. Most of the time, the developers' cognitive process involves identifying, understanding, and analyzing code written by other developers. Therefore, it is not rare in cases where the human understanding of a particular code snippet diverges from the machine interpretation, leading to an erroneous conclusion about the code snippet outcome in a future execution (GOPSTEIN *et al.*, 2017).

Previous studies showed that code comprehension is the most dominant activity in the development process, consuming about 58% of the total time spent (MINELLI *et al.*, 2015; XIA *et al.*, 2018). When programmers are involved in high comprehension effort, they navigate and make edits at a significantly slower rate (RAHMAN, 2018). Moreover, code reviewers often do not understand the change being reviewed or its context (EBERT *et al.*, 2017). In this circumstance, confusing code impacts code comprehension and, also, the development process. In addition, developers tend to understand specific code structures more quickly than other ones (more challenging), e.g., `for` loops take more time to be understood than sequences of `ifs` (AJAMI *et al.*, 2019). Also, some programming practices affect the code readability (SANTOS; GEROSA, 2018).

Gopstein *et al.* (2017) identified code patterns responsible for creating confusion in developers. These patterns were named Atoms of Confusion (ACs), considered the smallest piece of code that can confuse programmers, hindering the correct understanding of the source code under consideration. 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 compared to codes with equivalent

functions that do not include these atoms, often leading to unexpected results (GOPSTEIN *et al.*, 2017). This seminal work showed that these patterns could significantly impact correctness, and the time and effort of program understanding. Despite the importance shown by Gopstein *et al.* (2017), ACs studies are still few. For example, the impact caused by ACs in other programming languages, such as Java, and how this phenomenon is spread to software systems calls for further investigation.

Different contexts of use of programming languages in software development can influence the way ACs manifest themselves in code. Aspects such as technical characteristics and programming styles, often formalized in guidelines, used by the developer communities of each language have a direct impact on the occurrence of ACs. Atoms defined in a given language may not apply or may need to be adapted for another language. Similarly, developer communities may encourage programming practices that insert atoms into the source code. Therefore, expanding the study of ACs to programming languages other than C and C++ is important for understanding how this phenomenon occurs and also its consequences in more specific contexts.

Based on the study of Gopstein *et al.* (2017), Langhout e Aniche (2021) 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.* (2017), resulting in a list of 14 reproducible ACs in Java (LANGHOUT; ANICHE, 2021). Inspired by the studies of Langhout e Aniche (2021) and Gopstein *et al.* (2017), we decided to investigate whether the phenomena found in C and C++ systems also occurred in the Java ecosystem.

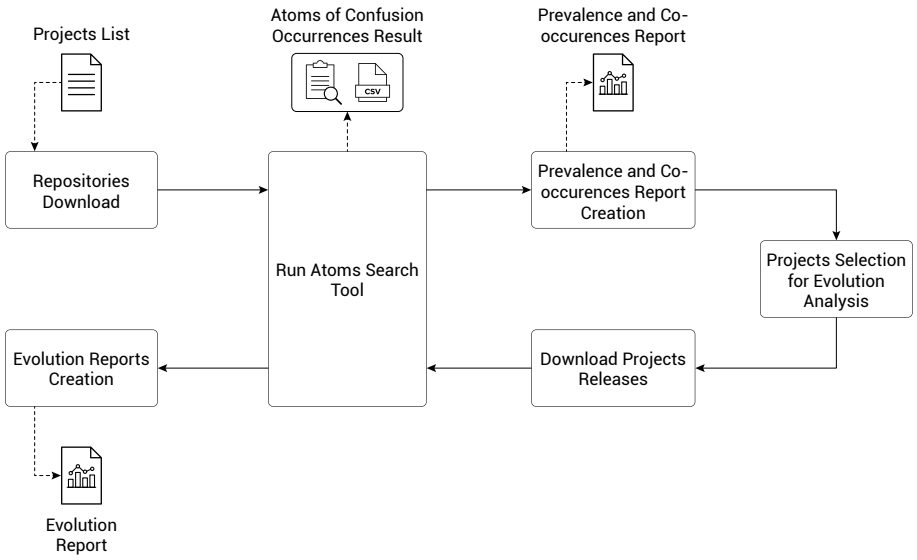
1.2 Study Methodology

Our study investigates the prevalence of ACs 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 address this phenomenon adequately.

To achieve this goal, we developed a tool for searching ACs in Java source code: BOHR. This tool works on top of Spoon[1], which is a Java source code analysis and transformation library. With BOHR, 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 six other traditional libraries: Gson,

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

Figure 1 – Prevalence and Evolution study workflow



Source: The author.

Hamcrest, Jsoup, JUnit, Mockito, and X-Stream). It is worth pointing out that these libraries are developed under a rigorous quality assurance process, with high test coverage.

Figure 1 shows our study's main phases. First, we selected 27 long-lived Java libraries (see Section 4.1) 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.

1.3 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 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 ACs, 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 and 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., Does a class implementing mathematical operations is more likely to have ACs?). At this first moment, we chose not to evaluate co-occurrence at the method level or by proximity, since this would add more complexity to the study.

RQ3. *How the prevalence of Atoms of Confusion evolve over time in long-lived Java libraries*?

We studied the evolution of the occurrences of the ACs in 24 libraries from the set of 27 selected. The goal is to evaluate how long ACs prevail 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.

1.4 Goals and Contributions

There are some studies about Atoms of Confusion including definitions and confusion in code comprehension (GOPSTEIN *et al.*, 2017; CASTOR, 2018), prevalence of ACs in C/C++ software projects (GOPSTEIN *et al.*, 2018), ACs in context of Java programming language and its impacts on code comprehension (LANGHOUT; ANICHE, 2021), etc. Nevertheless, there could not be found researches that focus on prevalence of ACs in Java software projects yet.

This dissertation aims to provide a first insight into the prevalence of ACs in Java programs. We also intent to provide a tool to automatic ACs detection in Java programs, which generate CSV reports and also give an API for developing new custom finders to capture new ACs and any new code patterns. In addition, we provide a double-checked gold standard AC dataset that was used to validate this tool accuracy. Hence, we expected to achieve the following contributions:

- a prevalence, co-occurrence and evolution analysis of ACs in long-lived Java libraries.
- a tool for automatic ACs detection in Java programs.
- an API for developing new custom ACs finders.
- a double-checked gold standard ACs dataset.

During our analysis, 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 9 types of ACs, while Gson library had 7 types. We found 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. In the evolution analysis we showed the 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 (This library has undergone structural changes, its main class module has decreased in terms of LoC, which explains this condition). 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.

1.5 Document Organization

The remainder of this work is organized as follows. Chapter 2 discuss the background and related work. In Chapter 3, we present BOHR, the tool we built to support our study. Next, in Chapter 4, we describe the study results and implications. Finally, Chapter 5 presents the final considerations and proposals for further investigation.

2 BACKGROUND AND RELATED WORK

This chapter discusses general concepts and definitions related to Code Comprehension and Atoms of Confusion, providing the theoretical foundation for this study. This chapter is organized as follows: Section 2.1 presents the works about Code Comprehension; Section 2.2 discusses the concepts of Atoms of Confusion; in Section 2.3, we discuss the concepts of Atoms of Confusion in the context of Java programming language; and, finally, Section 2.4 concludes the chapter.

2.1 Code Comprehension

Code comprehension is the process in which developers constantly gain knowledge about a system by exploring and researching software artifacts, reading source code and system documentation. This acquired knowledge helps support other software engineering activities, such as bug fixing, enhancement, reuse, and documentation (XIA *et al.*, 2018). Previous studies have shown that the activity of code comprehension is essential and time-consuming in the development and maintenance of systems. Minelli *et al.* (2015) analyzed over 700 hours of work by 18 developers, 7 professionals and 11 graduate students, and concluded that about 70% of the programming time is spent understanding the system. Xia *et al.* (2018) went further and conducted a large-scale study with 78 professional developers, for a total of 3,145 hours worked on 7 real industry projects, and concluded that about 58% of the time was spent on code comprehension activities.

In another work, Ajami *et al.* (2019) used an experimental platform modeled as an online game-like environment to measure how quickly and accurately 220 professional programmers can interpret code snippets with similar functionality, but different structures. The snippets that took longer to understand or produced more errors were considered more difficult. This study showed that some code structures are more difficult to understand than others, for example, *for* loops are significantly more difficult than *ifs*, and countdown loops are a bit more complicated to understand than count up loops. This demonstrates how different ways of writing code affect the process of understanding the program.

Santos e Gerosa (2018) evaluated the impact of a set of programming practices in Java on code readability. For each coding practice, a pair of code snippets was defined in a way that one snippet adhered to the practice and the other violated it. The results showed that of the 11

coding practices evaluated, 8 affected the perceived readability of the surveyed developers, while for 3 of these practices there was insufficient evidence to claim that they affect code readability.

In another previous work, Rahman (2018) related the program comprehension effort to programming activities. This research observed that when programmers are involved in a high comprehension effort they navigate and make edits at a significantly slower rate. Hence, the study showed the impact that confusing code can have on programmers' productivity in software development.

In the context of code review Ebert *et al.* (2017) presented a *framework* to identify confusion in comments left in code by reviewers. This study showed that reviewers often do not understand the change being reviewed and its context. Furthermore, the research also showed that confusion can be reasonably well-identified by reviewers. These results highlight a negative impact caused by confusing codes, which are susceptible to different interpretations on the code review process.

Finally, using an electrophysiological approach, Yeh *et al.* (2017) used an electroencephalogram device to record the brain activity of individuals during the program comprehension process. The results indicated a higher average in magnitude when solving more confusing codes when compared to non-confusing codes. It was also found that there was no difference in the mean magnitudes between resolutions of the same type of code snippet. Therefore, there is evidence of a relationship between magnitude and cognitive workload, also that understanding confusing codes requires more brain activity.

2.2 Atoms of Confusion

In (GOPSTEIN *et al.*, 2017) was 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 understanding of the source code and can hinder the development process, leading programmers to take longer in programming activities and to make mistakes in maintenance tasks, introducing bugs in the system.

To assist researchers interested in the study of Atoms of Confusion, Castor (2018) defined an AC as a code pattern that is:

- precisely identified;
- likely to confuse;

- replaceable by a functionally equivalent code pattern that is less likely to cause confusion; and
- indivisible.

In addition to providing this structured definition of atoms of confusion, Castor (2018) identified some sources of confusion that exist in the original atoms catalog and presented a preliminary atom catalog for Swift.

Gopstein *et al.* (2017) pointed out 15 ACs that cause significant confusion when present in source code. In complementary work was performed a study of prevalence of confusing code of the most popular and influential open source C/C++ software projects. In this work was observed that the 15 known types of ACs occur millions of times in programs like the Linux kernel and GCC, appearing on average once every 23 lines. It has also been noted 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 (GOPSTEIN *et al.*, 2018).

In a more recent work, was 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 (GOPSTEIN *et al.*, 2020).

In another recent research was evaluated code interpretations with and without ACs using an eye tracker. From an aggregate perspective, a 43.02% increase in time required to understand code correctly 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 (OLIVEIRA *et al.*, 2020).

2.3 Atoms of Confusion in Java

Based on the study of Gopstein *et al.* (2017), Langhout e Aniche (2021) 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.*, 2017), 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 AC, 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 (LANGHOUT; ANICHE, 2021). Table 1 presents the list of the 10 ACs based on that work, their respective Java translations, and the code with the confusion removed.

The study of Langhout e Aniche (2021) in the context of the Java ecosystem motivated our research to investigate the incidence of ACs using an automated search tool developed by us.

Table 1 – Atoms of Confusion in Java adapted from (LANGHOUT; ANICHE, 2021)

Atom Name	**Snippet with AC**	**Snippet without AC**
Infix Operator Precedence	`2 + 4 * 2;`	`2 + (4 * 2);`
Post-Increment/Decrement	`a = b++;`	`a = b;` `b += 1;`
Pre-Increment/Decrement	`a = ++b;`	`b += 1;` `a = b;`
Conditional Operator	`b = a == 3 ? 2 : 1;`	`if(a == 3){b = 2;}` `else{b = 1;}`
Arithmetic as Logic	`(a - 3) * (b - 4) != 0`	`a != 3 && b != 4`

Logic as Control Flow	`a == ++a > 0 || ++b > 0`	`if(!(a + 1 > 0)) {`
		`  b += 1;}`
		`a += 1`

Change of Literal Encoding	`a = 013;`	`a = Integer.parseInt("13", 8);`
Omitted Curly Braces	`if(a) f1(); f2();`	`if(a){ f1(); } f2();`
Type Conversion	`a = (int) 1.99f;`	`a = (int) Math.floor(1.99f);`

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

2.4 Conclusion

In this chapter, we presented the concept of Code Comprehension, its goals, and how this process takes place in software development. The concept of Atoms of Confusion were clarified and discussed, as well as their impacts on software development, especially on the activity of code comprehension. The Table 2 presents a summary of the background and related works showed in this chapter.

Table 2 – Projects used in the Precision and Recall Evaluation

Research	**Programming Language**	**Method**	**Main Findings**
Gopstein *et al.* (2017)	C/C++	Manual code analysis	Introduction of the concept of Atom of Confusion and proposition of its types
Gopstein *et al.* (2018)	C/C++	Prevalence analysis through automatic code analysis	ACs are prevalent, buggy, confusing, unique and waning
Castor (2018)	Swift	Manual code analysis	Structured definition of AC and a preliminary catalog of ACs for Swift.
Gopstein *et al.* (2020)	C/C++	Focus on the 'how' and 'why' of programmer misunderstanding through interviews	Correct answers about the results of a code do not guarantee that there was no confusion in its comprehension process
Oliveira *et al.* (2020)	C/C++	Eye-tracking camera to detect the visual attention of the participants while solving the tasks	Increase in time required to understand code correctly and in number of gaze transitions in code snippets with ACs. Code regions with atoms receives most of the eye attention
Langhout (2020)	Java	translation of ACs from C/C++ to Java and evaluation of the perceptions and impacts of ACs on novice developers through surveys	Developers are more likely to make misunderstandings in 7 of the 14 ACs studied and they reported that versions containing ACs are more confusing and less readable in 10 of the 14 ACs investigated.

Source: the author.

Although there are some studies on Atoms of Confusion, no research was found that focused on the prevalence of ACs in Java programs. Our study aims to contribute to this research area by investigating the prevalence of ACs in Java software projects and providing a tool to automatic ACs detection in Java programs.

3 BOHR TOOL

This chapter presents BOHR, a tool for searching ACs in Java source code. Section 3.1 shows an overview of this tool. Section 3.2 presents BOHR's architecture. Section 3.3 details all the AC types covered and each case of AC occurrence detected by BOHR. Section 3.4 presents the process of dataset creation to BOHR evaluation. Next, Section 3.5 describes BOHR accuracy evaluation process. Finally, Section 3.6 concludes the chapter.

3.1 The Atoms of Confusion Hunter

To answer our research questions, we developed a tool to search ACs in programs written in Java automatically. BOHR aims to check for the presence of ACs in Java classes and provides information about them. The information found can be exported in a report as a Comma Separated Values (CSV) file. This CSV file shows the code snippets that contain ACs, their types, class names and the lines in which they were found. The image 2 shows a CSV file sample of the generated report.

Figure 2 – CSV File Sample

Class	Atom	Snippet	Line
com.google.gson.internal.sql.SqlTimeTypeAdapter	Conditional Operator	value == null ? null : format.format(value)	67
com.google.gson.JsonObject	Conditional Operator	value == null ? JsonNull.INSTANCE : value	58
com.google.gson.stream.JsonReader	Logic as Control Flow	((pos < limit) || fillBuffer(1)) && (buffer[pos] == '>')	524
com.google.gson.stream.JsonReader	Logic as Control Flow	if (((pos + i) >= limit) && (!fillBuffer(i + 1))) { return PEEKED_NONE; }	621
com.google.gson.stream.JsonReader	Logic as Control Flow	if ((c != keyword.charAt(i)) && (c != keywordUpper.charAt(i))) { return PEEKED_NONE; }	625
com.google.gson.stream.JsonReader	Type Conversion	(long) asDouble	964
com.google.gson.stream.JsonReader	Type Conversion	(int) peekedLong	1169
com.google.gson.stream.JsonReader	Type Conversion	(int) asDouble	1201
com.google.gson.internal.bind.util.ISO8601Utils	Post Increment Decrement	value.charAt(i++)	309
com.google.gson.internal.bind.util.ISO8601Utils	Post Increment Decrement	value.charAt(i++)	316
com.google.gson.internal.bind.util.ISO8601Utils	Conditional Operator	(millis) ? ".sss".length() : 0	73
com.google.gson.internal.bind.util.ISO8601Utils	Conditional Operator	(tz.getRawOffset() == 0) ? "Z".length() : "+hh:mm".length()	74
com.google.gson.internal.LinkedHashTreeMap	Infix Operator Precedence	Integer.highestOneBit(targetSize) * 2 - 1	682
com.google.gson.internal.LinkedHashTreeMap	Post Increment Decrement	(size++) > threshold	194
com.google.gson.stream.JsonWriter	Infix Operator Precedence	size > 1 || size == 1 && stack[size - 1] != NONEMPTY_DOCUMENT	557
com.google.gson.stream.JsonWriter	Post Increment Decrement	stack[stackSize++]	358
com.google.gson.stream.JsonWriter	Logic as Control Flow	(!lenient) && (Double.isNaN(value) || Double.isInfinite(value))	494
com.google.gson.internal.bind.TypeAdapters	Type Conversion	(byte) intValue	183

Source: The author.

The use of the CSV file type allows the data to be exploited by various types of software compatible with this format (spreadsheet-generating software, software APIs, database systems, business intelligence softwares, etc.). Thus, researchers can manipulate the generated data in any way they wish. From this report it is possible, for example, to check the prevalence of ACs and extract data that can substantiate different analyses of the impacts caused by their presence.

Currently, BOHR is able to detect 10 of the 14 types of ACs presented by Langhout e Aniche (2021): Infix Operator Precedence (IOP), Post-Increment/Decrement (Post-Inc/Dec),

Pre-Increment/Decrement (Pre-Inc/Dec), Conditional Operator (CO), Arithmetic as Logic (AaL), Logic as Control Flow (LaCF), Change of Literal Encoding (CoLE), Omitted Curly Braces (OCB), Type Conversion (TC) and Repurposed Variables (RV). The detection of *Repurposed Variables* 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 in (LANGHOUT, 2020). All rules defined for each AC covered by BOHR are presented in detail on Section 3.3 and also in the BOHR's source code repository available in the following link: https://github.com/wendellmfm/bohr.

Langhout (2020) argues that are some kind of ACs that can be more easily avoided in the Java context, such as *Remove Indentation*, *Indentation*, and *Dead, Unreachable, Repeated*. For example, developers can avoid the *Remove Indentation* and *Indentation* atoms 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 ACs through warning messages. Also, the *Constant Variable* atom was not shown to be statistically significant, both in (GOPSTEIN *et al.*, 2017) and (LANGHOUT, 2020). Therefore, we decided not to include the detection of these four ACs in this version of our search tool.

3.2 BOHR Main Components

BOHR was developed using Spoon, an open-source library to analyze, rewrite, transform and transpile Java source code. Spoon parses source files to build a well-designed AST (*Abstract Syntax Tree*), a tree-based representation of source code, with powerful analysis and transformation API. It fully supports modern Java versions up to Java 16 (PAWLAK *et al.*, 2015).

BOHR performs inspection of Java systems from the directory path of the code repository files. From this informed path, processors responsible for the detection of specific atoms, called *Finders*, act in the search for ACs. This tool allows the analyst to select the desired *Finders* at each inspection execution. Data about the atoms found is stored and can be exported in a report as a CSV file.

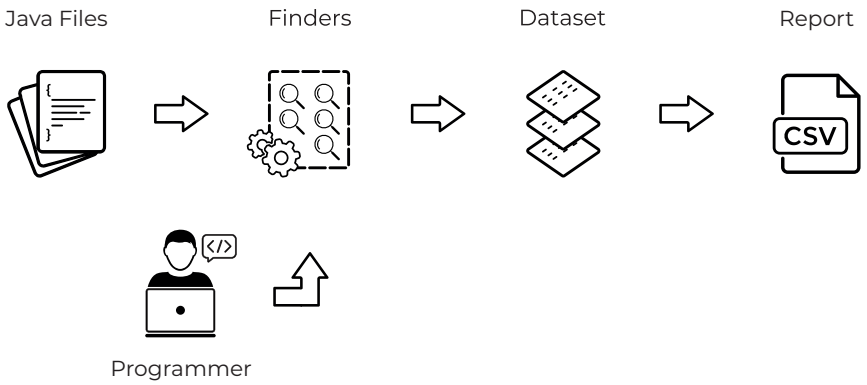
Since studies on ACs is still recent, new confused code patterns may emerge, so BOHR also includes an API that allows the researchers to implement their own *Finders* for the

detection of new code patterns. In this way, our tool can extend its capability and cover new types of atoms that may arise.

To validate the ACs detection rules, we built a set of automated unit tests, using the JUnit testing framework (JUNIT, 2021). This set of tests was based on the code examples from the work of Langhout (2020) and, in addition, through in-depth studies of the occurrences of each of the ACs types, we refined and added more test cases throughout our study.

Figure 3 presents an overview of the tool workflow.

Figure 3 – BOHR's Workflow Overview



Source: The author.

Code snippet 1 shows BOHR usage. The *findAoC* method of the *BohrAPI* class returns a collection of *AoCSuite* that contains information about ACs occurrences. This method receives as parameters the source code path, a boolean indicating whether the report will be generated and the report's destination path.

Source code 1 – BOHR usage

```
1 Collection<AoCSuite> suiteList = BohrAPI.findAoC("C:\\Project\\src\\main", true, "REPORT_PATH");
```

The source code 2 presents the implementation of a *Finder* responsible for identifying the *Conditional Operator* atom. The *process* method receives a Java Class as a parameter, where it first checks if this element is valid, then a filter is set to capture conditional expressions using the "?" operator (ternary expressions). Finally, all the ternary conditional expressions contained in the analyzed Class are stored in a *Dataset* object.

Programmers can create their own custom *Finders* by extending the abstract class `AbstractProcessor` from Spoon (PAWLAK *et al.*, 2015) and implementing its *process* method. Then, to use the newly implemented *Finder*, the *findAoC* method of the *BohrAPI* class is used.

This method can receive as parameter a list of *Strings*, where each item in this list specifies the qualified name of the class that implements the newly *Finder*. The source code 3 shows the use of a custom Finder.

Source code 2 – Conditional Operator Finder

```
public class ConditionalOperatorFinder extends AbstractProcessor<CtClass<?>> {

  public void process(CtClass<?> element) {
    if (Util.isValid(element)) {
    String className = element.getQualifiedName();

      TypeFilter<CtConditional<?>> filter = new TypeFilter<CtConditional<?>>(CtConditional.class);
      for (CtConditional<?> condOpr : element.getElements(filter)) {
        if ((condOpr.getParent() != null) {
            if((condOpr.getParent() instanceof CtAssignment)
            || condOpr.getParent() instanceof CtLocalVariable)) {
          int lineNumber = condOpr.getParent().getPosition().getLine();
          String snippet = condOpr.getParent().prettyprint();
          Dataset.store(className, new AoCInfo(AoC.CoO, lineNumber, snippet));
          }
        }
      }
    }
  }
}
```

Source code 3 – Custom Finder usage

```
String[] customFinders = new String[] { "br.ufc.mdcc.CustomFinder" };
Collection<AoCSuite> suiteList = BohrAPI.findAoC("C:\\Project\\src\\main", customFinders, false, null);
```

3.3 Atoms of Confusion and BOHR Detection Rules

This section presents all types of ACs covered, detailing each case of AC occurrence detected by BOHR. Pseudo codes describing the Finders' behaviors are also presented, as well as their examples in Java code.

3.3.1 Infix Operator Precedence

This AC occurs when more than one type of binary operator is used in a code instruction. The confusion is caused by a misunderstanding of the order of execution of these operators. The multiplication, division and modulus operators have execution precedence over

the addition and subtraction operators, as well as the `and` operator (`&&`) have execution precedence over the `or` operator (`||`). The transformed variant of this atom includes parentheses around the operations to clarify the order of these operations, making the code expression more readable and easier to understand.

BOHR detects this atom when arithmetic and logical expressions do not have parentheses around operations with higher precedence.

Algorithm 1: Infix Operator Precedence Finder pseudo code

```
if operation is binary then
    if operation is of type arithmetic then
        if type of binary operator is equal to multiplication, division or modulo then
            if binary operator has another binary operator as its parent then
                if type of binary operator parent is equal to addition or subtraction then
                    if the binary operation is not enclosed in parentheses and it is not the
                     case of a concatenation of strings then
                        store Infix Operator Precedence occurrence
                    end
                end
            end
        end
    end
    if operation is of type logical then
        if type of binary operator is equal to && or || then
            if binary operator has another binary operator as its parent then
                if type of binary operator is equal to && and its parent is equal to || or
                 type of binary operator is equal to || and its parent is equal to && then
                    if binary operation is not enclosed in parentheses then
                        store Infix Operator Precedence occurrence
                    end
                end
            end
        end
    end
end
```

Source code 4 – Arithmetic expressions

```
int a = 2 + 2 * 4;
```

Source code 5 – Logical expression

```
if(a || b && c) {
  System.out.println("true");
} else {
  System.out.println("false");
}
```

3.3.2 Pre-Increment/Decrement

This AC consists on the use of the pre-increment/decrement unary operators (++ and - -). The pre-increment/decrement unary operator increments/decrements the variable associated with and returns the result of the expression. The lack of familiarity with this operator generates doubts about its operation, which can confuse. Moreover, another possible confusion is due to the pre-increment/decrement operator that can be confused with the post-increment/decrement operator, which returns the result of the operation only returns the variable's value.

BOHR detects occurrences of this atom when the pre-increment/decrement operator appears in:

1. Variable assignments.
2. Binary operations.
3. Parameter in method invocations.
4. An index on reading arrays.
5. Returns of methods.

Algorithm 2: Pre Increment Decrement pseudo code

```
if operation is unary then
    if unary operation has parent then
        if parent of a unary operation is a binary operation, an assignment, a local
          variable, a method invocation, an array access or a return statement then
            if unary operator is type of pre increment or decrement then
                store Pre Increment Decrement occurrence
            end
        end
    end
end
```

Source code 6 – Variable assignments

```
int a = 2;
int b = ++a;
System.out.println(a +" "+ b);
```

Source code 7 – Variable assignments

```
int a = 2;
int b = --a;
System.out.println(a +" "+ b);
```

Source code 8 – Binary operations

```
int a = 1;
int b = 3 + --a;
System.out.println(b);
```

Source code 9 – Binary operations

```
int a = 0;
if(++a == 0) {
  System.out.println("true");
}
```

Source code 10 – Parameter in method invocations

```
System.out.println(method(++a));
```

Source code 11 – An index on reading arrays

```
int a = 1;
int[] array = {0, 1, 2, 3, 4};
System.out.println(array[++a]);
```

Source code 12 – Returns of methods

```
private static int method() {
    int a = 1;
    return ++a;
}
```

3.3.3 Post-Increment/Decrement

The *Post-Increment/Decrement* atom is, in a sense, complementary to the *Pre-Increment/Decrement* explained earlier. Likewise, this atom is also based on the use of a unary operator, but in this case, it is the post increment/decrement operator. As explained before, the difference is that instead of the result of the expression the original value of the variable is returned. The confusion caused by this atom is also due to a lack of familiarity with how it works, as well as the fact that the post-increment/decrement operator can be confused with the pre-increment/decrement operator.

BOHR detects occurrences of this atom when the post increment/decrement operator appears in:

1. Variable assignments.
2. Binary operations.
3. Parameter in method invocations.
4. An index on reading arrays.
5. Returns of methods.

Algorithm 3: Post Increment Decrement pseudo code

```
if operation is unary then
    if unary operation has parent then
        if parent of a unary operation is a binary operation, an assignment, a local
          variable, a method invocation, an array access or a return statement then
            if unary operator is type of post increment or decrement then
                store Post Increment Decrement occurrence
            end
        end
    end
end
```

Source code 13 – Variable assignments

```
int a = 2;
int b = a++;
System.out.println(a +" "+ b);
```

Source code 14 – Variable assignments

```
int a = 2;
int b = a--;
System.out.println(a +" "+ b);
```

Source code 15 – Binary operations

```
int a = 1;
int b = 3 + a--;
System.out.println(b);
```

Source code 16 – Binary operations

```
int a = 0;
if(a++ == 0) {
  System.out.println("true");
}
```

Source code 17 – Parameter in method invocations

```
System.out.println(method(a++));
```

Source code 18 – An index on reading arrays

```
int a = 1;
int[] array = {0, 1, 2, 3, 4};
System.out.println(array[a++]);
```

Source code 19 – Returns of methods

```
private static int method() {
    int a = 1;
    return a++;
}
```

3.3.4 *Conditional Operator*

This atom is based on the use of the ternary operator (`?:`), which is a shortened form of the `if-then-else` code structure. The syntax of the ternary operator can cause confusion in developers who are not familiar with this structure.

BOHR captures all occurrences of the ternary operator as *Condition Operator* atom.

Algorithm 4: Conditional Operator pseudo code

if *operation is of type conditional* **then**
  store Conditional Operator occurrence
end

Source code 20 – Conditional operator on assignment

```
int a = 4;
int b = a == 3 ? 2 : 1;
System.out.println(b);
```

Source code 21 – Conditional operator on return statement

```
public int method() {
int a = 1;
return a == 3 ? 2 : 1;
}
```

3.3.5 *Arithmetic as Logic*

Consists of using arithmetic operators instead of logical operators. In Java, the result of the operation must generate a boolean value, for that we must explicitly add a comparison to the expression. Thus, in order to have equivalence between arithmetic and logical operations, the BOHR detects arithmetic expressions, involving variables, composed as follows:

1. Multiplication operations equal to or not equal to zero.
2. Addition or subtraction operations equal to or not equal to zero.

Algorithm 5: Arithmetic as Logic pseudo code

```
if operation is binary then
    if operation has equals or not equals operator then
        if left or right hand operand equals to "0" then
            if operation has arithmetic as logic expression then
                store Arithmetic as Logic occurrence
            end
        end
    end
end
```

Source code 22 – Multiplication operation equal to zero

```
if(a * b == 0) {
  System.out.println("true");
} else {
  System.out.println("false");
}
```

Source code 23 – Multiplication operation not equal to zero

```
int a = 8;
if((a - 3) * (7 - a) != 0) {
  System.out.println("true");
} else {
  System.out.println("false");
}
```

Source code 24 – Addition operations equal to zero

```
int a = 5;
if(a + 5 == 0) {
  System.out.println("true");
} else {
  System.out.println("false");
}
```

Source code 25 – Subtraction operations not equal to zero

```
int a = 5;
if(a - 5 != 0) {
  System.out.println("true");
} else {
  System.out.println("false");
}
```

3.3.6 Logic as Control Flow

This atom is based on the "lazy" behavior of the logical operators `&&` and `||`, where depending on the value of the left-hand side expression, the right-hand side expression may or may not be executed. In this way, these logical operators can also be used as conditionals.

BOHR considers the code snippet to be *Logic as Control Flow* when there is, on the right-hand side of the logic operation, some operation that indicates or may indicate some change of values of system variables. BOHR considers the following types of these operations:

1. Unary operators.
2. Method invocations.
3. Variable assignments.

Therefore, if any of these types of instructions occur on the right-hand side of a logical operation of the `&&` and `||` operators, the BOHR will understand it as an occurrence of Logic as Control Flow atom.

Algorithm 6: Logic as Control Flow pseudo code

```
if operation is binary then
    if type of binary operator is equal to && or || then
        if operation has unary operation, assignment or method invocation in right hand
         operand then
            store Logic as Control Flow occurrence
        end
    end
end
```

Source code 26 – Unary operator

```
int a = 1;
int b = 2;
if(a > 0 && ++b > 2) {
  a = a - 1;
  b = b + 2;
}
System.out.println(a +" "+ b);
```

Source code 27 – Method invocations

```
int a = 1;
if(a > 0 && method()) {
  a = a * 2;
}
System.out.println(a);
```

Source code 28 – Variable assignments

```
int a = 1;
int b = 4;
if(a > 0 && (b = 3) != 4) {
  System.out.println(b);
}
```

3.3.7 Repurposed Variables

Usually, in programming, variables are created for specific purposes. This atom consists of reusing an existing variable for another purpose in the program. In this way, when a variable is used in different roles across the life time of a program, its correct understanding may be compromised. Due to the complexity of detecting this atom because of the difficulty of automatically inferring the semantics of the variables, our tool detects this atom only in two case pointed out in (LANGHOUT, 2020), in these cases we define well defined structures of their occurrences. This atom is detected when:

1. An index check of an object of type Array appears as a stop condition in a loop, `for` or `while`, and also, in this same loop, this same object has a write operation.
2. In a nested `for`, when the same update variable is used in both loops, the inner and the outer.

Algorithm 7: Repurposed Variables pseudo code

```
if there is a for or a while loop then
    if there is an array reading in loop expression then
        if there is an array writing in loop body then
            store Logic as Control Flow occurrence
        end
    end
end
if there is a for loop that has another for loop as parent then
    if there is an array reading in loop expression then
        if the same variable is initialized in the outer for as update variable of the inner
         for then
            store Repurposed Variables occurrence
        end
    end
end
```

Source code 29 – An index check of an array as stop condition and write operation

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

Source code 30 – Inner and outer For loops with the same update variable.

```
int a = 3;
for(int i = 0; i < 2; i++) {
  for(int j = 0; i < 2; i++) {
    a = 4 * i + j;
  }
}
System.out.println(a);
```

3.3.8 Change of Literal Encoding

To represent numerical values in programs we tend to use decimal format and, occasionally, binary, hexadecimal or octal for specific cases. Even if they contain the same number, these different representations can create confusion in understanding the values. In this

sense, this atom is based on the use of different formats to represent numeric values that can cause confusion in the understanding of the program.

BOHR detects cases of this atom when:

1. A literal numeric value beginning with zero is located, indicating that it is an octal representation.
2. A binary bitwise operation, `bitand`, `bitor` or `bitxor` (`&`, `|` or `^`), where at least one of the operands is a literal and is in decimal format.

Algorithm 8: Change of Literal Encoding pseudo code

```
if there is a literal in variable assignment then
    if the literal is in octal base then
        store Change of Literal Encoding occurrence
    end
end
if operation is binary then
    if type of binary operator is equal to bitand, bitor or bitxor (&,| or ^) then
        if there is at least one operand in decimal base then
            store Change of Literal Encoding occurrence
        end
    end
end
```

Source code 31 – Literal numeric value beginning with zero

```
int a = 013;
System.out.println(a);
```

Source code 32 – Binary bitwise operation with literal in decimal format.

```
int a = 11 & 32;
System.out.println(a);
```

3.3.9 Omitted Curly Braces

Consists of the confusion that can be caused due to a lack of clarity in the separation of code blocks. In Java, the code structures `if-then-else`, `for` and `while` have a flexibility in their declarations when they only have one code statement, in this case the use of braces

for encapsulating this statement is optional. In this sense, not using braces in these cases can imply unclear separation between blocks of code and cause confusion in the understanding of the program.

BOHR detects *Omitted Curly Braces* when the `if-then-else`, `for` and `while` structures satisfy all the following conditions:

1. Have a single code statement;
2. Do not have curly braces encapsulating this statement; and
3. The next statement of the program must appear on the same line as the instruction belonging to that structure (`if-then-else`, `for` or `while`).

Algorithm 9: Omitted Curly Braces pseudo code

```
if code block then
    if code block belongs to an if then
        call IfOmittedCurlyBracesDetection
    end
    if code block belongs to an else then
        if there is just one else statement then
            if does not have curly braces encapsulating then
                if next statement is on the same line then
                    if does not have else block then
                        store Omitted Curly Braces occurrence
                    end
                    else
                        call IfOmittedCurlyBracesDetection
                    end
                end
            end
        end
    end
end
Function IfOmittedCurlyBracesDetection():
    if there is just one if statement then
        if does not have curly braces encapsulating then
            if next statement is on the same line then
                store Omitted Curly Braces occurrence
            end
        end
    end
End Function
```

Source code 33 – Next statement on the same line as a single If statement

```
int a = 2;
if(a <= 4) a++; a++;
System.out.println(a);
```

Source code 34 – Next statement on the same line as a single If statement

```
int a = 2;
if(a <= 4)
    a++; a++;
System.out.println(a);
```

Source code 35 – Next statement on the same line as a single For statement

```
int a = 2;
for(int i = 0; i <= 4; i++) a++;
    a++;
System.out.println(a);
```

Source code 36 – Next statement on the same line as a single For statement

```
int a = 2;
for(int i = 0; i <= 4; i++)
    a++; a++;
System.out.println(a);
```

Source code 37 – Next statement on the same line as a single While statement

```
int a = 2;
while(a < 4) a++; a++;
System.out.println(a);
```

Source code 38 – Next statement on the same line as a single While statement

```
int a = 2;
while(a < 4)
  a++; a++;
System.out.println(a);
```

3.3.10 Type Conversion

This atom occurs when there is a conversion from a larger data type to a smaller type, this type of conversion is known as *Narrowing Conversion*, for example, the conversion of a data type from `float` to `int`. In these cases there can be losses of precision, causing results that may be unexpected by the programmer. In Java we have several situations where there can be losses of precision in data conversions between primitive types.

BOHR detects all possible cases of *Type Conversion* between primitive types in Java. Here is the list of possible situations:

1. `short` to `byte` or `char`.
2. `char` to `byte` or `short`.
3. `int` to `byte`, `short` or `char`.
4. `long` to `byte`, `short`, `char` or `int`.
5. `float` to `byte`, `short`, `char`, `int` or `long`.
6. `double` to `byte`, `short`, `char`, `int`, `long` or `float`.

Conversions involving the primitive type `char` add another layer of complexity, since it is the only type that is unsigned, and converting characters to numbers is not intuitive.

Using APIs in explicit Narrowing conversions, such as `java.lang.Math` and `java.lang.Character`, for example, tends to make the code more readable. Also in this type of conversion, the use of the % (modulo) operator can indicate a treatment on the data to be converted. This is achieved by modulo operation of the data of larger type, data to be converted, with the number of possible values that the data of smaller type can represent, 256 numbers in the case of byte, for example.

To be considered a *Type Conversion* the data conversion must be explicit and:

1. not present method invocation in this process. A method invocation may indicate a possible use of APIs or some treatment for handling the conversion;
2. not present a modulo operation of the data to be converted; and
3. if the data to be converted is a literal and its value is outside the possible range of representations of the converted type.

Algorithm 10: Type Conversion pseudo code

```
if there is cast expression then
    if there is narrowing conversion then
        if does not have method invocation and modulo operation on converted data then
            store Type Conversion occurrence
        end
    end
    if expression is literal then
        if there is narrowing conversion and the literal value is out of representation
         range then
            store Type Conversion occurrence
        end
    end
end
```

Source code 39 – Short to byte

```
short a = 288;
byte b = (byte) a;
System.out.println(b);
```

Source code 40 – Char to short

```
char a = '4';
short b = (short) a;
System.out.println(b);
```

Source code 41 – Int to char

```
int a = 4;
char b = (char) a;
System.out.println(b);
```

Source code 42 – Long to int

```
long a = 2147483648L;
int b = (int) a;
System.out.println(b);
```

Source code 43 – Float to long

```
float a = 1.99f;
long b = (long) a;
System.out.println(b);
```

Source code 44 – Double to long

```
double a = 1.99;
long b = (long) a;
System.out.println(b);
```

Source code 45 – Int to char

```
public char method(int v) {
  System.out.println(v);
  return (char) v;
}
```

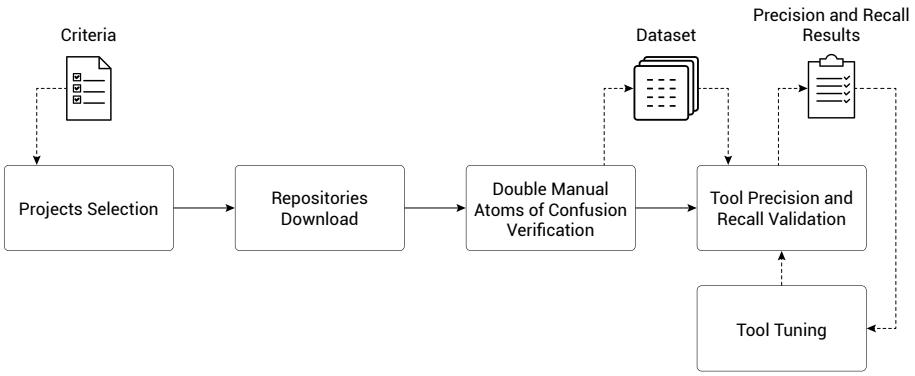
Source code 46 – Float to int

```
public int method(float v) {
  System.out.println(v);
  return (int) v;
}
```

3.4 BOHR's Precision and Recall 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 4 shows the workflow of this evaluation. The dataset creation was divided in two steps: projects selection and manual inspection.

Figure 4 – Precision and Recall study workflow



Source: The author.

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 BOHR. In this sense, observing the classification of Pinto *et al.* (2015) (PINTO *et al.*, 2015), we choose small projects (i.e., LoC $\leq$ 20,000). To check criterion (v), we ran BOHR on several candidate projects that met the previous

four criteria and and checked whether the desired condition was fulfilled.

Four projects satisfied the inclusion criteria: FastUtil[1], Moshi[2], Jimfs[3], and uCrop[4]. As none of these projects contained the *Repurposed Variables* and *Arithmetic as Logic* atoms, we created a sample project, by extracting Java files containing these two ACs types from the Guava (version 31.0.1)[5] and Redisson (version 3.6.16)[6] projects. Table 3 presents the selected projects and their respective versions used in the evaluation.

Table 3 – 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

Source: the author.

We manually checked all Java files in these projects' main source code package, excluding the test files, to search for ACs. 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 occurrences 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 the line number in which it was located. Table 4 shows the numbers of ACs by type per project we found.

3.5 Precision and Recall Evaluation

We ran BOHR 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 detected by BOHR were also tagged manually. Therefore, there was no identification error. However, some issues with the tool recall appeared. For example, BOHR did not was able to identify all occurrences of the *Type of Conversion* and

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

Table 4 – ACs occurrences in Precision and Recall Evaluation dataset

AC Type	FastUtil	Moshi	Jimfs	uCrop	Sample
Infix Operator Precedence	8	5	5	31	1
Pre-Increment/Decrement	3	-	8	-	-
Post-Increment/Decrement	20	10	4	8	-
Conditional Operator	18	74	31	31	1
Omitted Curly Braces	1	-	-	-	-
Logic as Control Flow	12	65	58	28	7
Arithmetic as Logic	-	-	-	-	4
Change of Literal Encoding	-	2	9	-	2
Type Conversion	24	4	3	13	3
Repurposed Variables	-	-	-	-	2

Source: the author.

Infix Operator Precedence atoms.

In the case of *Type of Conversion*, the recall issue occurred due to unsupported types of conversions (i.e., literals, unary operations, and binary operations). In the *Infix Operator Precedence* 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 subsection 3.3.

3.6 Conclusion

In this chapter, we presented the tool proposed in this work, BOHR - The Atoms of Confusion Hunter, which aims to obtain the prevalence information of ACs in Java systems. We described the structures of the BOHR, their functioning and their features. The list of ACs covered by our tool was detailed, as well as the cases in which BOHR detects ACs occurrences.

The precision and recall evaluations of BOHR were also detailed. In addition, we presented the double-check gold standard dataset used in this evaluation process.

4 PREVALENCE STUDY

Once the effectiveness of BOHR was confirmed, we started the study on the prevalence of ACs in long-lived Java libraries. This chapter describes the prevalence study of our research using the tool detailed in the previous chapter. Section 4.1 describes the selection of long-lived Java libraries analyzed. Section 4.2 details materials, procedures and methods of this study. Section 4.3 shows prevalence results. Section 4.4 presents co-occurrences findings. Section 4.5 shows evolution analysis results. Section 4.6 discusses the study results and their implications. Finally, Section 4.7 concludes the chapter.

4.1 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.* (2021). This set comprises 27 libraries, 21 libraries from the Apache Commons[1] 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 (PAWLAK *et al.*, 2015).

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 ACs in these projects provides a good indication of how this phenomenon occurs and evolves in Java projects. Table 5 summarizes the selected libraries for this study.

4.2 Materials, Procedures and Methods

To conduct this study we developed BOHR, the tool described in the previous chapter, to automatically search for ACs in Java programs. Thus, we were able to check for the presence of ACs in Java files and provide information about them. The BOHR's source code repository available in the following link: https://github.com/wendellmfm/bohr.

To validate our tool we manually built a double-checked gold standard dataset to assess its precision and recall. This validation process was detailed in Section 3.4 and Section 3.5. The dataset generated in this experiment is available in: https://zenodo.org/record/7065842#

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

.YxvJh2zMJD8

Finally, to conduct the prevalence study of ACs, we downloaded a total of 455 releases of long-lived Java libraries for this study. These downloads were performed manually from official repositories. After that, we built a script to extract the main directory of code from the projects, excluding auxiliary and test modules. Then, we built another script that used BOHR to analyze the releases. As a result, we obtained ACs prevalence reports that supported our study. Reports in CSV and XLSX files are available in: https://zenodo.org/record/7065882#.YxvQzmzMJD8. All release versions analyzed in this study are available in Appendix A.

4.3 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 5 shows the results for all selected libraries, number of classes containing ACs, number of ACs found, and number of types present in each library. $11,643$ ACs were found, with an average ≈ 2.1 ACs per class and a rate ≈ 1.00 AC per 40 lines of code.

Figure 5 brings two interesting information: (1) the distribution of AC types over all occurrences of ACs (left) and (2) the prevalence of AC types across libraries (right). On one hand, regarding (1), the *Logic as Control Flow*, *Infix Operator Precedence*, and *Conditional Operator* represent together more than 86% of all occurrences. In contrast, *Pre-Increment/Decrement*, *Change of Literal Encoding*, *Repurposed Variables*, and *Arithmetic as Logic* combined represent less than 2,50% of all ACs occurrences. On the other hand, concerning (2), we found that *Conditional Operator* and *Logic as Control Flow* had 100% of prevalence. The *Pre-Increment/Decrement* and *Infix Operator Precedence* reached 81,48% and 70,37% of prevalence, respectively. *Type Conversion* and *Post-Increment/Decrement* achieved average prevalence rate, with 66,67% and 59,26%, respectively. The *Arithmetic as Logic* and *Repurposed Variables* reached both 11,11% of prevalence, the lowest rate.

Figure 6 shows the absolute number of occurrences of each AC type per library. This figure cross the information we first presented separately in Figure 5. 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

Table 5 – Projects Information and Prevalence Results

Library	Version	LoC	Classes	Classes w/ACs	ACs	Types
BCEL	6.5.0	31,686	391	76	322	7
BeanUtils	1.9.4	11,644	111	36	174	5
CLI	1.5.0	2,151	23	12	84	3
Codec	1.15	9,313	72	37	436	7
Collections	4.4	28,955	326	96	565	6
Compress	1.2.1	44,730	359	174	1,155	9
Configuration	2.7	28,011	260	92	342	6
DBCP	2.9.0	14,454	66	31	127	2
DbUtils	1.7	3,074	46	19	29	2
Digester	3.2	9,917	168	39	94	5
Email	1.5	2,815	23	12	50	5
Exec	1.3	1,757	32	11	38	4
FileUpload	1.4	2,425	39	7	26	5
Functor	1.0	5,861	158	111	495	3
IO	2.11.0	14,024	180	77	358	7
Lang	3.12.0	29,745	215	80	880	8
Math	3.6.1	100,364	990	390	4,174	9
Net	3.8.0	20,199	212	86	389	6
Pool	2.11.1	5,905	49	16	80	5
Proxy	1.0	2,072	43	10	15	4
Validator	1.7	7,619	64	41	167	5
Gson	2.8.9	8,342	77	33	263	7
Hamcrest	2.2	3,505	80	9	17	3
Jsoup	1.14.3	13,714	73	39	323	6
JUnit	5.8.2	30,977	645	133	284	4
Mockito	4.3.0	20,298	467	87	249	5
X-Stream	1.4.19	21,859	361	164	507	6

Source: the author.

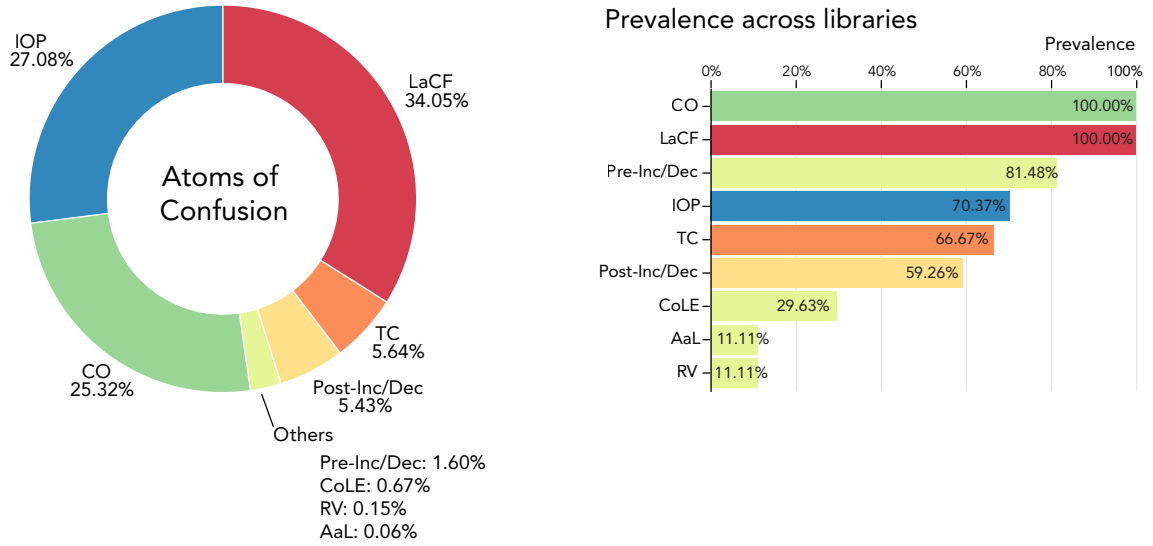
Repurposed Variables achieved the lowest rate in both.

Figure 7 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[2] 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

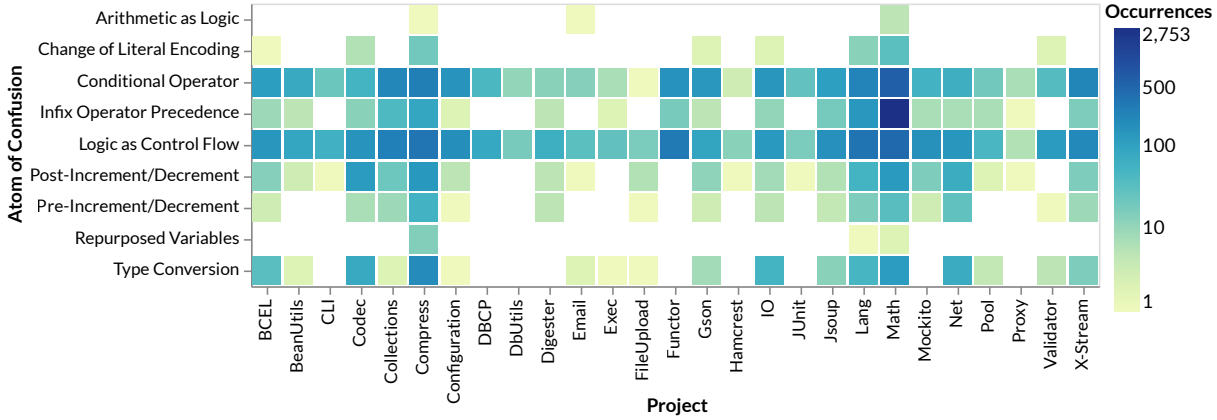
[2] We computed only Pearson's correlation in this study.

Figure 5 – Summary of ACs' prevalence. Left: Distribution of AC types over all the occurrences of ACs. Right: Prevalence of AC types across the studied libraries.



Source: the author.

Figure 6 – The absolute number of occurrences of ACs per library.

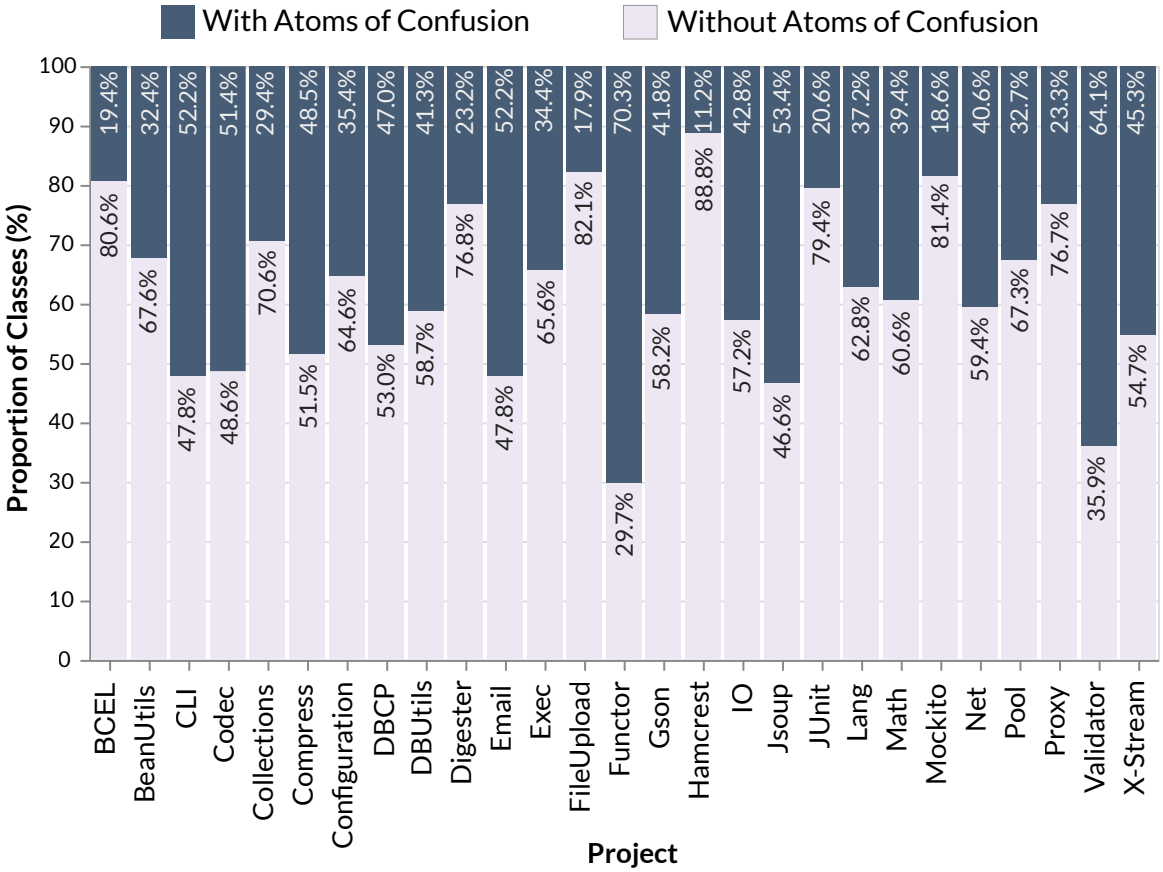


Source: the author.

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 LoC (100.364). It also contains the largest number of ACs types found (9), tied with the Compress library, the second-largest library in terms of LoC (44.730). The Proxy library possesses the fewest number of ACs occurrences (15), and is the second smallest library in terms of LoC (2.072). Finally, the DBCP and DbUtils libraries have the fewest number of distinct ACs types (2).

Table 6 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 atom was also quite frequent in the C and C++ projects analyzed by Gopstein *et al.* (2018). The wide usage of *Logic as Control Flow* may be due to the short form of expressing a conditional structure, rather than using the

Figure 7 – Proportion of classes with and without ACs per library.



Source: the author.

`if-else` statement (LANGHOUT, 2020).

Infix Operator Precedence was the second most common atom, 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 atom, the Lang library, had only 138 occurrences. This AC was also the second most common in the C and C++ projects studied in (GOPSTEIN *et al.*, 2018). *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 (GOPSTEIN *et al.*, 2018).

The *Conditional Operator* was the third most frequent atom, with 2.874 occurrences, as also observed in (GOPSTEIN *et al.*, 2018). In that study, it was also one of the most common atom. The *Conditional Operator* is also encouraged by the software engineering community. Kernighan e Pike (1999) state that the use of the ternary operator (`<condition> ? <expression> : <expression>`) is good for short expressions. In a similar prevalence

Table 6 – Prevalence Results by AC Type

Library	IOP	Pre-Inc/Dec	Post-Inc/Dec	CO	LaCF	AaL	CoLE	TC	RV
BCEL	9	3	14	117	145	-	1	33	-
BeanUtils	5	-	3	77	87	-	-	2	-
CLI	-	-	1	23	60	-	-	-	-
Codec	13	7	123	53	157	-	6	77	-
Collections	42	9	22	220	270	-	-	2	-
Compress	88	56	139	279	360	1	20	197	15
Configuration	2	1	5	152	181	-	-	1	-
DBCP	-	-	-	46	81	-	-	-	-
DbUtils	-	-	-	11	18	-	-	-	-
Digester	5	5	5	13	66	-	-	-	-
Email	-	-	1	14	32	1	-	2	-
Exec	2	-	-	7	28	-	-	1	-
FileUpload	-	1	6	1	17	-	-	1	-
Functor	18	-	-	163	314	-	-	-	-
IO	11	5	8	146	133	-	2	53	-
Lang	138	16	52	242	369	-	13	49	1
Math	2,753	34	129	616	484	5	32	119	2
Net	7	29	69	64	147	-	-	73	-
Pool	7	-	2	20	47	-	-	4	-
Proxy	1	-	1	7	6	-	-	-	-
Validator	-	1	-	37	122	-	2	5	-
Gson	5	3	12	142	91	-	2	8	-
Hamcrest	-	-	1	3	13	-	-	-	-
Jsoup	19	4	6	111	170	-	-	13	-
JUnit	3	-	1	100	180	-	-	-	-
Mockito	7	3	16	56	167	-	-	-	-
X-Stream	16	9	16	231	218	-	-	17	-

Source: the author.

study, the authors omitted the *Conditional Operator* atom in their experiment because of its high number of occurrences in practice (MEDEIROS *et al.*, 2019).

We did not find the *Omitted Curly Braces* in any studied library, in contrast to what was observed in (GOPSTEIN *et al.*, 2018). In that study, the *Omitted Curly Braces* was the most common atom 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. (GOPSTEIN *et al.*, 2018) 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*" (KERNEL, 2022). 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*" (APACHE, 2022). 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*" (GOOGLE, 2022). Thus, the different development contexts of the Java and C/C++ languages may explain the big difference in the results of *Omitted Curly Braces* prevalence.

The *Repurposed Variables* had a significant frequency in (GOPSTEIN *et al.*, 2018) in contrast to what we observed in our study, in which this atom was rare, with just 18 occurrences. The *Arithmetic as Logic* was also rare in our study with only 7 occurrences, but this AC was not included in the study of Gopstein *et al.* (2018).

4.4 RQ2. To what extent do different types of Atoms of Confusion co-occur, at the class level, in long-lived Java libraries?

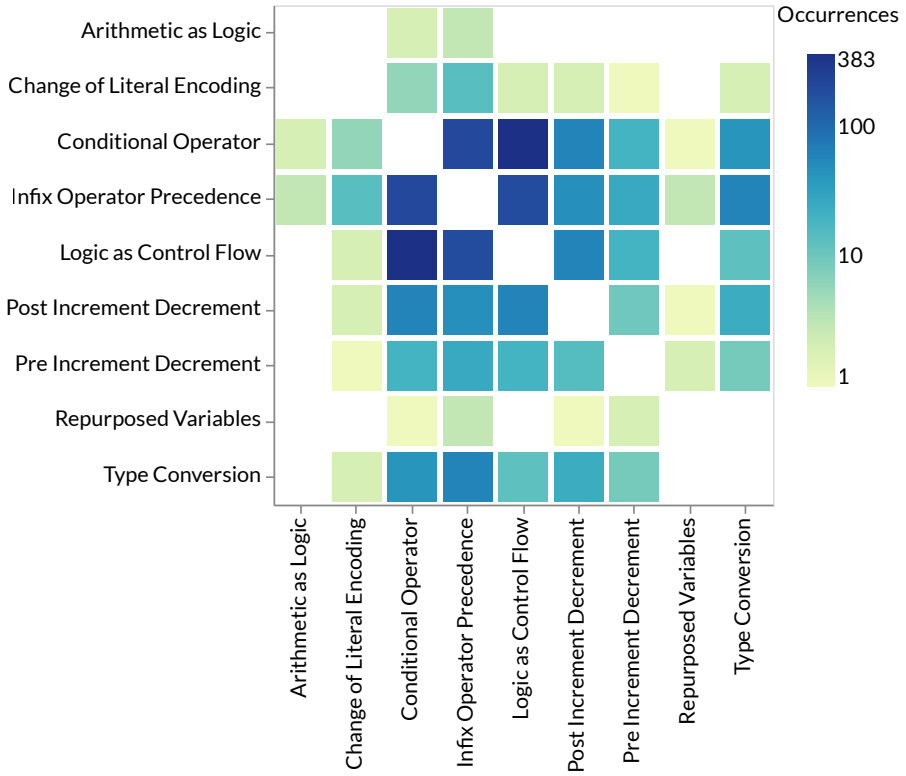
We measured the co-occurrence of ACs in the same class. Our goal was to observe tendencies for certain atoms to occur together in the same Java file. Figure 8 presents a co-occurrence matrix of ACs at the class level for all libraries. We learned that the *Conditional Operator, Logic as Control Flow* and *Infix Operator Precedence* are more likely to co-occur in the same class. These results confirm a trend pointed out by the RQ1 results, as these three ACs that co-occur more frequently at the class level are also the three most common atoms in the libraries studied.

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

We found 7 different AC types 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.

Figure 8 – Atoms of confusion co-occurrence matrix for all libraries.



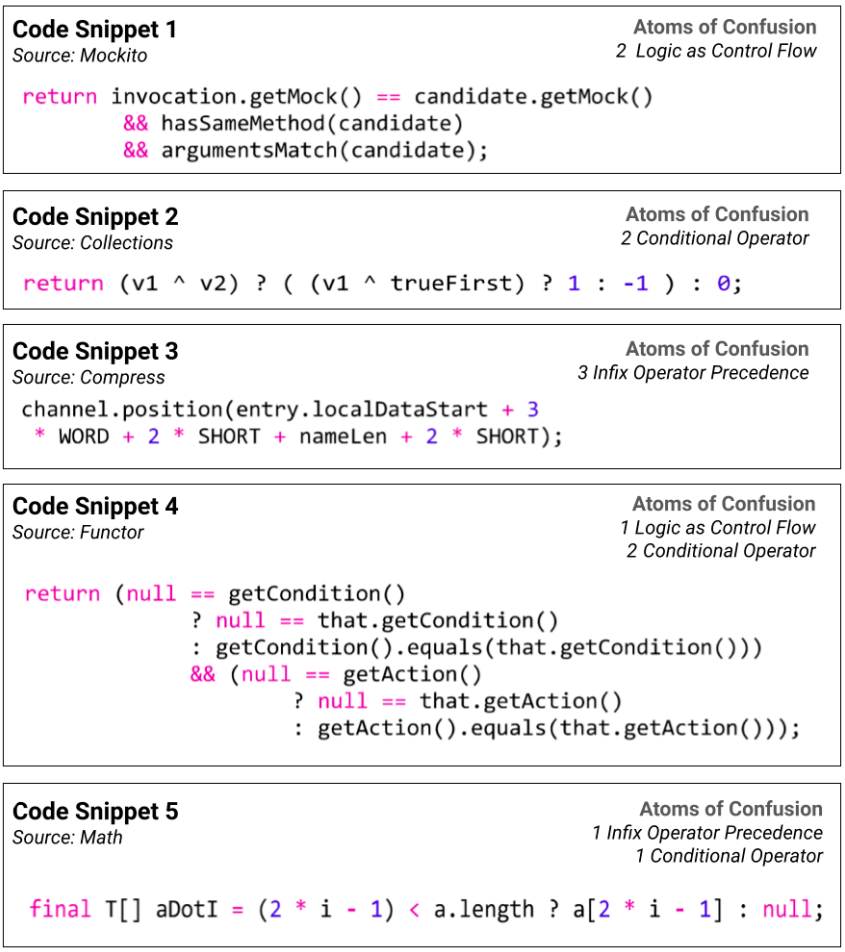
Source: the author.

Figure 9 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* atoms. The code snippet 2, from the Collections library, shows two *Conditional Operator* atoms. The code snippet 3 has three *Infix Operator Precedence* atoms in the Compress library. The code snippet 4, which our tool found in the Functor project, contains one *Logic as Control Flow* and one *Conditional Operator*. Finally, in the code snippet 5, extracted from Math, we have one *Infix Operator Precedence* and one *Conditional Operator*.

4.5 RQ3. How the prevalence of Atoms of Confusion evolve over time 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 7 shows data from the first and last versions of this 24 libraries. The analysis covered a total of 455 releases. While Gson and Jsoup libraries had the highest number of

Figure 9 – Atoms of confusion co-occurrence code snippets.



Source: the author.

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, which can be explained by the 41.55% reduction in its 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 4.009 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 LoC. 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 7 – Evolution of ACs and LoC in the 24 projects

	First Release				Last Release				Variation		
Library	**Version**	**Classes**	**LoC**	**ACs**	**Version**	**Classes**	**LoC**	**ACs**	**Classes**	**LoC**	**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%▲

Source: the author.

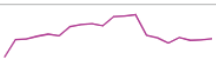
Table 8 shows the evolution of the ratio number of ACs to the number of LoC over time and, also, the spread of ACs in library classes over time. The libraries indicated different behaviors for the two variables observed. In 11 libraries, the ratio of ACs to LoC decreased. In the case of the Codec and 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. 4 of them, BeanUtils, CLI, Gson, and X-Stream, showed an increase of ACs to LoC 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 8 libraries. For example, the CLI, Compress, Exec, and Jsoup projects had both increases and decreases in this variable over time.

In 15 libraries the percentage of Java classes with AC increased in the observed period. 4 libraries had more than 100% increases: BeanUtils, IO, Gson, and X-Stream. Moreover, in the last two libraries the percentage grew practically with each new version. On the other hand, in 5 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%.

Table 8 – Evolution of ACs/LoC 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%		

Source: the author.

4.6 Results Discussion

As we mentioned, GOPSTEIN *et al.* 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 (GOPSTEIN *et al.*, 2017). From the Java code patterns of these atoms (LANGHOUT; ANICHE, 2021), our study found 11,404 occurrences in the 27 projects studied.

Our results showed that 23 of the 27 analyzed libraries had atoms in more than 20% of their classes (RQ1). There was a presence of ACs 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, BOHR, 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, *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 10 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 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.

The detection of *Logic as Control Flow* atom proved particularly challenging because it is very common to use methods just to read values into the program, such as 'get' and 'equals' methods. We noticed that these methods proved to be very common, appearing as right hand operand in logic operations in the analyzed systems. Since these methods do not change values in the program, they only read the values to be tested in the logic operation, it is up to the programmer to observe whether the right-hand operand method is actually performing a write operation by changing the values in its body. Therefore, accurate and automatic identification of

this type of atom remains a challenge.

The *Repurposed Variable* detection is strictly related to the semantics of program variables. Detecting the change of purpose of variables in a program during its life cycle is not a trivial task. The detection of this AC by our tool occurs only in two cases pointed out in (LANGHOUT, 2020), where there is a well-defined code structure that enables its detection. Hence, automatically inferring this change in purpose continues to be a challenge.

4.6.1 Implications for Researchers

The presence of ACs in long-lived Java libraries grows over time. This phenomenon needs further investigation into why developers insert ACs into 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.

4.6.2 Implications for Practitioners

As we stated before, previous work has shown that confusing code impacts code comprehension and, hence, the development process. When programmers are involved in high comprehension effort, they navigate and make edits at a significantly slower rate (RAHMAN, 2018). Code reviewers often do not understand the change being reviewed or its context (EBERT *et al.*, 2017). Also in the context of software development, programmers 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` (AJAMI *et al.*, 2019). As well as some programming practices also affect the code readability (SANTOS; GEROSA, 2018). In the context of ACs was observed that there is a strong relationship between ACs and bug fix commits and also pointed out that atoms tend to be more commented in source code (GOPSTEIN *et al.*, 2018). Hence, it is important to disseminate this knowledge among developers, alerting them to the presence of these ACs in code.

Since ACs can lead to problems related to code understanding during software development, maintenance and evolution, such as an increase in effort and time to understand the source codes of programs, as well as possible misunderstandings in this process, this work

can help programmers avoid writing source code that contains atoms and also help promote refactoring actions aimed at removing ACs from the source code of systems.

In addition, developers can use BOHR 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.

4.7 Conclusion

This chapter presented the results of the prevalence, co-occurrence and evolution analyses of ACs in long-lived Java libraries. 455 releases of this Java libraries were downloaded, according to selection criteria described, to evolution analysis. The methods and procedures for conducting this research were detailed, as well as the materials used and provided.

5 CONCLUSIONS

This chapter presents the conclusions of this study. Section 5.1 presents main contributions of this work. Section 5.2 discusses the threats to validity. Section 5.3 shows our final considerations. Finally, Section 5.4 presents proposals for further investigations.

5.1 Main Contributions

The main contributions of this work are summarized below:

- **BOHR**: a tool to automatically search ACs in programs written in Java available in: https://github.com/wendellmfm/bohr
- **Double-checked gold standard dataset of ACs**: a dataset that enables further validations of new tools for ACs identification. Available in: https://zenodo.org/record/7065842#.YxvJh2zMJD8
- **Prevalence study**: prevalence, co-occurrence and evolution analyses of ACs in long-lived Java programs. Results of this study available in: https://zenodo.org/record/7065882#.YxvQzmzMJD8

In addition, 2 papers were published along the development of this work:

- W. Mendes, W. Viana, and L. Rocha, "BOHR-Uma Ferramenta para a Identificação de Átomos de Confusão em Códigos Java" **Anais do IX Workshop de Visualização, Evolução e Manutenção de Software. SBC**, 2021.
- W. Mendes, O. Pinheiro, E. Santos, W. Viana and L. Rocha, "Dazed and Confused: Studying the Prevalence of Atoms of Confusion in Long-Lived Java Libraries" 2021 **IEEE International Conference on Software Maintenance and Evolution (ICSME)**, 2022.

5.2 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.* (2012) (WOHLIN *et al.*, 2012).

5.2.1 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 (GOPSTEIN *et al.*, 2018) (MEDEIROS *et al.*, 2019) (FILHO *et al.*, 2019). Thus, we use the count, frequency and proportion of ACs in the studied softwares as metrics.

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

5.2.3 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.

5.2.4 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.

5.3 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 ACs 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 development, 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 ACs in Java-based systems. This dataset enables the validation of new tools for atoms identification, while our tool, BOHR, enables programmers to find and remove ACs from Java source code.

5.4 Future Work

In future work, we intend to study the impact of ACs on software quality attributes, such as bug occurrence, technical debt, code complexity, and maintainability effort. We also plan to improve the co-occurrence analysis of ACs and refine our atom detection tool.

We aim to analyze the relationship of ACs to the occurrence of bugs, looking at the correlation of removing ACs in bug-fix commits, comparing to other types of commits. We also intend to investigate the presence of ACs as an indicator of technical debt, since these atoms can compromise the legibility of the code and consequently its maintainability.

Confusing code can lead to misunderstandings and increase effort and time in software development. ACs can cause a recursive effort where the presence of these patterns can lead to more code modifications. We aim to evaluate the impact on development time and system maintenance effort. In addition, we intend to investigate whether codes with ACs have a higher complexity (cyclomatic complexity or other readability-related metric) than codes free of ACs, in other words, whether the presence of ACs adds complexity to the code.

In the context of the ACs co-occurrence analysis, we intend to further investigate the relationships between atoms that occur together, checking aspects such as proximity and scope of occurrences (e.g., co-occurrence at method level) and investigating the role of programmers in this phenomenon (e.g., do classes changed by more programmers tend to have more co-occurrences?).

Also, in the context of our developed tool, some ACs detections can be improved, such as *Logic as Control Flow* and *Repurposed Variables*. In the future we plan to revisit the studies of these atoms and improve their detections.

REFERENCES

AJAMI, S.; WOODBRIDGE, Y.; FEITELSON, D. G. Syntax, predicates, idioms—what really affects code complexity? **Empirical Software Engineering**, Springer, v. 24, n. 1, p. 287–328, 2019.

APACHE. **Coding Standards**. 2022. https://commons.apache.org/proper/commons-net/code-standards.html. Acesso em: 14 Jan. 2022.

BENNETT, K.; RAJLICH, V.; WILDE, N. Software evolution and the staged model of the software lifecycle. In: ZELKOWITZ, M. V. (Ed.). Elsevier, 2002, (Advances in Computers, v. 56). p. 1–54. Disponível em: https://www.sciencedirect.com/science/article/pii/S0065245802800031.

CASTOR, F. Identifying confusing code in swift programs. In: **Proceedings of the VI CBSoft Workshop on Visualization, Evolution, and Maintenance. ACM**. [*S. l.*: *s. n.*], 2018.

EBERT, F.; CASTOR, F.; NOVIELLI, N.; SEREBRENIK, A. Confusion detection in code reviews. In: **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S. l.*: *s. n.*], 2017. p. 549–553.

FILHO, F. G. de A.; MARTINS, A. D. F.; VINUTO, T. d. S.; MONTEIRO, J. M.; SOUSA, I. P. de; MACHADO, J. de C.; ROCHA, L. S. Prevalence of bad smells in pl/sql projects. In: **Proceedings of the 27th International Conference on Program Comprehension**. IEEE Press, 2019. (ICPC '19), p. 116–121. Disponível em: https://doi.org/10.1109/ICPC.2019.00025.

GOOGLE. **Google Java Style Guide**. 2022. https://google.github.io/styleguide/javaguide.html#s4.1-braces. Acesso em: 14 Jan. 2022.

GOPSTEIN, D.; FAYARD, A.-L.; APEL, S.; CAPPOS, J. Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion. In: . New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 605–616. ISBN 9781450370431. Disponível em: https://doi.org/10.1145/3368089.3409714.

GOPSTEIN, D.; IANNACONE, J.; YAN, Y.; DELONG, L.; ZHUANG, Y.; YEH, M. K.-C.; CAPPOS, J. Understanding misunderstandings in source code. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 129–139. ISBN 9781450351058. Disponível em: https://doi.org/10.1145/3106237.3106264.

GOPSTEIN, D.; ZHOU, H. H.; FRANKL, P.; CAPPOS, J. Prevalence of confusing code in software projects: Atoms of confusion in the wild. In: . New York, NY, USA: Association for Computing Machinery, 2018. (MSR '18), p. 281–291. ISBN 9781450357166. Disponível em: https://doi.org/10.1145/3196398.3196432.

JUNIT. **JUnit 5**. 2021. https://junit.org/junit5/. Acesso em: 1 Mar. 2021.

KERNEL. **Linux kernel coding style**. 2022. https://www.kernel.org/doc/html/v4.10/process/coding-style.html. Acesso em: 14 Jan. 2022.

KERNIGHAN, B. W.; PIKE, R. **The practice of programming**. [*S. l.*]: Addison-Wesley Professional, 1999.

LANGHOUT, C. **Investigating the Perception and Effects of Misunderstandings in Java Code**. Dissertação (Mestrado) – Delft University of Technology, 2020.

LANGHOUT, C.; ANICHE, M. **Atoms of Confusion in Java**. 2021.

LIMA, L.; ROCHA, L.; BEZERRA, C. I. M.; PAIXAO, M. Assessing exception handling testing practices in open-source libraries. **Empirical Software Engineering**, v. 26, 09 2021.

MEDEIROS, F.; LIMA, G.; AMARAL, G.; APEL, S.; KäSTNER, C.; RIBEIRO, M.; GHEYI, R. An investigation of misunderstanding code patterns in c open-source software projects. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 24, n. 4, p. 1693–1726, ago. 2019. ISSN 1382-3256. Disponível em: https://doi.org/10.1007/s10664-018-9666-x.

MINELLI, R.; MOCCI, A.; LANZA, M. I know what you did last summer - an investigation of how developers spend their time. In: **2015 IEEE 23rd International Conference on Program Comprehension**. [*S. l.*: *s. n.*], 2015. p. 25–35.

OLIVEIRA, B. de; RIBEIRO, M.; COSTA, J. A. S. da; GHEYI, R.; AMARAL, G.; MELLO, R. de; OLIVEIRA, A.; GARCIA, A.; BONIFáCIO, R.; FONSECA, B. Atoms of confusion: The eyes do not lie. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 243–252. ISBN 9781450387538. Disponível em: https://doi.org/10.1145/3422392.3422437.

PAWLAK, R.; MONPERRUS, M.; PETITPREZ, N.; NOGUERA, C.; SEINTURIER, L. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. **Software: Practice and Experience**, Wiley-Blackwell, v. 46, p. 1155–1179, 2015. Disponível em: https://hal.archives-ouvertes.fr/hal-01078532/document.

PINTO, G.; TORRES, W.; FERNANDES, B.; CASTOR, F.; BARROS, R. S. A large-scale study on the usage of java's concurrent programming constructs. **Journal of Systems and Software**, v. 106, p. 59–81, 2015. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121215000849.

RAHMAN, A. Comprehension effort and programming activities: Related? or not related? In: **Proceedings of the 15th International Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2018. (MSR '18), p. 66–69. ISBN 9781450357166. Disponível em: https://doi.org/10.1145/3196398.3196470.

RUGABER, S. Program comprehension. **Encyclopedia of Computer Science and Technology**, v. 35, n. 20, p. 341–368, 1995.

SANTOS, R. M. a. dos; GEROSA, M. A. Impacts of coding practices on readability. In: **Proceedings of the 26th Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2018. (ICPC '18), p. 277–285. ISBN 9781450357142. Disponível em: https://doi.org/10.1145/3196321.3196342.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. [*S. l.*]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435.

XIA, X.; BAO, L.; LO, D.; XING, Z.; HASSAN, A. E.; LI, S. Measuring program comprehension: A large-scale field study with professionals. **IEEE Transactions on Software Engineering**, v. 44, n. 10, p. 951–976, 2018.

YEH, M.; GOPSTEIN, D.; YAN, Y.; ZHUANG, Y. Detecting and comparing brain activity in short program comprehension using eeg. In: . [*S. l.*: *s. n.*], 2017. p. 1–5.

APPENDIX A – PROJECTS VERSIONS

bcel	beanutils	cli	codec	collections	compress	configuration	dbcp	dbutils	digester	email	exec	fileupload	io	lang	math	net	pool	validator	gson	jsoup	junit	mockito	xstream
5.2	1.5	1.0	1.1	1.0	1.0	1.0	1.0	1.0	1.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0.0	1.0	1.0	1.0	0.1.1	4.12	2.25.0	0.2
6.0	1.6	1.1	1.2	2.0	1.1	1.1	1.1	1.1	1.6	1.1	1.0.1	1.1	1.1	1.0.1	1.1	1.1.0	1.0.1	1.0.1	1.1	0.1.2	4.13	2.26.0	0.3
6.1	1.6.1	1.2	1.3	2.1	1.2	1.2	1.2	1.2	1.7	1.2	1.1	1.1.1	1.2	2.0	1.2	1.2.0	1.1	1.0.2	1.1.1	0.2.1	4.13.1	2.27.0	0.4
6.2	1.7.0	1.3	1.4	2.1.1	1.3	1.3	1.2.1	1.3	1.8	1.3	1.2	1.2	1.3	2.1	2.0	1.2.1	1.2	1.1.3	1.2	0.2.2	4.13.2	2.28.0	0.5
6.3	1.8.0	1.3.1	1.5	3.0	1.4	1.4	1.2.2	1.4	1.8.1	1.3.1	1.3	1.2.1	1.3.1	2.2	2.1	1.2.2	1.3	1.1.4	1.2.1	0.3.1	5.0.0	3.0.0	0.6
6.3.1	1.8.1	1.4	1.6	3.1	1.4.1	1.5	1.3	1.5	2.0	1.3.2		1.2.2	1.3.2	2.3	2.2	1.3.0	1.4	1.2.0	1.2.2	1.1.1	5.0.1	3.1.0	1.0.1
6.4.0	1.8.2	1.5.0	1.7	3.2	1.5	1.6	1.4	1.6	2.1	1.3.3		1.3	1.4	2.4	3.0	1.4.0	1.5	1.3.0	1.2.3	1.2.1	5.0.2	3.10.0	1.0.2
6.4.1	1.8.3		1.8	3.2.1	1.6	1.7	2.0	1.7	3.0	1.4		1.3.1	2.0	2.5	3.1	1.4.1	1.5.1	1.3.1	1.3	1.2.2	5.0.3	3.11.0	1.0
6.5.0	1.9.0		1.9	3.2.2	1.7	1.8	2.0.1		3.1	1.5		1.3.2	2.0.1	2.6	3.1.1	2.0	1.5.2	1.4.0	1.4	1.2.3	5.1.0	3.12.0	1.1
	1.9.1		1.10	4.0	1.8	1.9	2.1		3.2			1.3.3	2.1	3.0	3.2	2.2	1.5.3	1.4.1	1.5	1.3.1	5.1.1	3.2.0	1.1.1
	1.9.2		1.11	4.1	1.8.1	2.0	2.1.1					1.4	2.2	3.0.1	3.3	3.0	1.5.4	1.5.0	1.6	1.3.2	5.2.0	3.3.0	1.1.2
	1.9.3		1.12	4.2	1.9	2.1	2.2.0						2.3	3.1	3.4	3.0.1	1.5.5	1.5.1	1.7	1.3.3	5.3.0	3.4.0	1.2
	1.9.4		1.13	4.3	1.10	2.1.1	2.3.0						2.4	3.2	3.4.1	3.1	1.5.6	1.6	1.7.1	1.4.1	5.3.1	3.5.0	1.2.2
			1.14	4.4	1.11	2.2	2.4.0						2.5	3.2.1	3.5	3.2	1.5.7	1.7	1.7.2	1.5.1	5.3.2	3.6.0	1.3
			1.15		1.12	2.3	2.5.0						2.6	3.3	3.6	3.3	1.6		2.0	1.5.2	5.4.0	3.7.0	1.3.1
					1.13	2.4	2.6.0						2.7	3.3.1	3.6.1	3.4	2.0		2.1	1.6.0	5.4.1	3.8.0	1.4
					1.14	2.5	2.7.0						2.8.0	3.3.2		3.5	2.1		2.2	1.6.1	5.4.2	3.9.0	1.4.2
					1.15	2.6	2.8.0						2.9.0	3.4		3.6	2.2		2.2.1	1.6.2	5.5.0	4.0.0	1.4.4
					1.16	2.7	2.9.0						2.10.0	3.5		3.7	2.3		2.2.2	1.6.3	5.5.1	4.1.0	1.4.5
					1.16.1								2.11.0	3.6		3.7.1	2.4		2.2.3	1.7.1	5.5.2	4.2.0	1.4.6
					1.17									3.7		3.7.2	2.4.1		2.2.4	1.7.2	5.6.1	4.3.0	1.4.7
					1.18									3.8		3.8.0	2.4.2		2.3	1.7.3	5.6.2		1.4.8
					1.19									3.8.1			2.4.3		2.3.1	1.8.2	5.6.3		1.4.9
					1.20									3.9			2.5.0		2.4	1.8.3	5.7.0		1.4.10
					1.21									3.10			2.6.0		2.5	1.9.1	5.7.1		1.4.11
														3.11			2.6.1		2.6	1.9.2	5.8.0		1.4.11.1
														3.12.0			2.6.2		2.6.1	1.10.1	5.8.2		1.4.12
																	2.7.0		2.6.2	1.10.2			1.4.13
																	2.8.0		2.7	1.10.3			1.4.14
																	2.8.1		2.8.0	1.11.1			1.4.15
																	2.9.0		2.8.1	1.11.2			1.4.17
																	2.10.0		2.8.2	1.11.3			1.4.18
																	2.11.0		2.8.3	1.12.1			1.4.19
																	2.11.1		2.8.4	1.12.2			
																			2.8.5	1.13.1			
																			2.8.6	1.14.1			
																			2.8.7	1.14.2			
																			2.8.8	1.14.3			
																			2.8.9				