



**FEDERAL UNIVERSITY OF CEARÁ**  
**CENTER OF SCIENCE**  
**DEPARTAMENT OF COMPUTER SCIENCE**  
**PROGRAM OF MASTER AND DOCTORATE IN COMPUTER SCIENCE**

**FRANCISCO OTON PINHEIRO NETO**

**HOW THEY RELATE AND LEAVE: UNDERSTANDING ATOMS OF CONFUSION IN  
OPEN-SOURCE JAVA PROJECTS**

**FORTALEZA**

**2024**

FRANCISCO OTON PINHEIRO NETO

HOW THEY RELATE AND LEAVE: UNDERSTANDING ATOMS OF CONFUSION IN  
OPEN-SOURCE JAVA PROJECTS

Dissertation submitted to the of 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

2024

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)

---

P719h Pinheiro Neto, Francisco Oton.

How they relate and leave : understanding atoms of confusion in open-source java projects / Francisco Oton Pinheiro Neto. – 2024.  
75 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2024.

Orientação: Prof. Dr. Windson Viana de Carvalho.

Coorientação: Prof. Dr. Lincoln Souza Rocha.

1. Program Comprehension. 2. Atoms of Confusion. 3. Empirical Study. 4. Data Mining. I. Título.

CDD 005

---

FRANCISCO OTON PINHEIRO NETO

HOW THEY RELATE AND LEAVE: UNDERSTANDING ATOMS OF CONFUSION IN  
OPEN-SOURCE JAVA PROJECTS

Dissertation submitted to the of 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: 29/02/2024

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. Matheus Henrique Esteves Paixão  
State University of Ceará (UECE)

To my mother, father, and sister, who have consistently been my pillars of support. They have not only provided me with the best education but also instilled in me a love for learning.

## **ACKNOWLEDGEMENTS**

Firstly, I thank God for guiding and strengthening me throughout this academic journey, granting me wisdom and perseverance. To my mother, Fátima, my boundless source of love and support, who courageously and dedicatedly assumed the role of educator after my father's passing, providing my sister and me with the best opportunities for growth and learning. To him, my father, Oton Filho, whose presence, albeit brief, left an eternal legacy in my life, always encouraging me to seek knowledge and celebrating every academic achievement with enthusiasm. To my sister, Natália, an example of determination and unconditional support, whose presence by my side was crucial in every moment and who often filled the void left by my father, playing a direct role in my education.

A special thanks to my girlfriend, Tyciane, for all the support, understanding, and companionship throughout this project, always advising and assisting me in making the best decisions during the project.

I extend my gratitude to my supervisors, Windson Viana and Lincoln Rocha, for all the support, valuable contributions, and for guiding me in the best possible way for this project. I thank my undergraduate friends, Pedro Paiva and Pedro Teixeira, who encouraged me to join the MDCC. Special thanks also to Wendell, my master's colleague student, who was instrumental in teaching me a great deal about my research topic and greatly facilitated the completion of this study.

## RESUMO

A compreensão de programa é essencial para aprimorar o entendimento e evitar erros no ciclo de vida do desenvolvimento de software. A confusão de código ocorre quando um desenvolvedor e o computador chegam a interpretações diferentes sobre o comportamento de um mesmo trecho de código. Tais trechos de código podem ser representados como pequenos e isolados padrões de código chamados Átomos de Confusão (ACs). Neste estudo, investigamos empiricamente os efeitos dos ACs no ciclo de vida de desenvolvimento de 21 projetos Java de código aberto. Construímos um *dataset* que relaciona mais de 8.000 *commits*, 4.000 *issues* e 7.000 ACs dos projetos em questão. Nossos resultados demonstraram uma correlação positiva entre o número de ACs e o número de *bugs* e melhorias relatados. Também investigamos mudanças em *commits*, buscando uma compreensão mais aprofundada do contexto no qual ACs são removidos. Como cada *commit* está vinculado a pelo menos uma *issue* relatada (por exemplo, *bug* e melhoria), conseguimos comparar a taxa de remoção de ACs em relação a cada tipo de *commit* e utilizá-la como um indicador para determinar se os ACs são provavelmente a causa por trás de uma *issue* reportada. Encontramos uma taxa mais elevada de remoção de ACs em *commits* de correção de *bugs* e melhorias do que em outros tipos de *commits* (tarefa, sub-tarefa, nova funcionalidade, desejo e teste) em 14 dos 19 projetos estudados, que tiveram ACs removidos em *commits*. Finalmente, para apoiar nossos resultados quantitativos, conduzimos uma análise qualitativa para melhor entender com que frequência átomos de confusão contribuíram para a ocorrência de *bugs* ou melhorias. Analisamos ACs removidos nesses tipos de *commits* com até dez linhas removidas, analisando o código-fonte, mensagens de cada *commit* envolvido, além do título, descrição e comentários das *issues* relacionadas no Jira. Em um universo de 8.641 *commits* de 21 projetos analisados, 391 removeram ACs. Dentre eles, 53 atenderam à condição para nossa análise qualitativa. Em 7 desses *commits*, 9 ACs removidos provavelmente contribuíram diretamente para a ocorrência de um *bug* ou melhoria. Até onde sabemos, nossa pesquisa é a primeira a investigar a conexão entre Átomos de Confusão e a ocorrência de *bugs* ou gatilhos para melhorias em projetos Java.

**Keywords:** compreensão de programa; átomos de confusão; estudo empírico; mineração de dados.

## ABSTRACT

Software comprehension is essential to improve understanding and avoid mistakes in the software development lifecycle. Code confusion occurs when a developer and the computer reach different interpretations about the behavior of the same piece of code. Such pieces of code can be represented as small and isolated code patterns called Atoms of Confusion (ACs). In this study, we empirically investigated the effects of ACs in the software development lifecycle of 21 open-source Java projects. We built a dataset linking more than 8,000 commits, 4,000 reported issues, and 7,000 ACs from the subject projects. Our findings showed a positive correlation between the number of ACs and the number of reported bugs and improvements. We also investigated changes in commits, looking forward to gathering a better understanding of in what context ACs are removed. As each commit is linked to at least one reported issue (e.g., bug and improvement), we were able to compare the ratio of ACs removal regarding each kind of commit and use it as a proxy to indicate whether ACs are likely to be the cause behind a reported issue. We found a higher ratio of removed ACs in bug-fix and improvement commits than in the other kinds of commits (task, sub-task, new feature, wish, and test) for 14 of the 19 studied projects, which had ACs removed in commits. Finally, to support our quantitative results, we conducted a qualitative analysis to understand better how often atoms of confusion contributed to the occurrence of a bug or improvement. We inspected ACs removed in these types of commits with up to ten lines removed, analyzing the source code, messages of each involved commit, and the title, description, and comments of related Jira issues. Out of a universe of 8,641 commits from 21 analyzed projects, 391 removed ACs. Among them, 53 met the condition for our qualitative analysis. In 7 of these commits, 9 removed ACs were likely to contribute directly to the occurrence of a bug or improvement. To the best of our knowledge, our research is the first to investigate the connection between Atoms of Confusion and the source of bugs or the cause of improvements in Java projects.

**Palavras-chave:** program comprehension; atoms of confusion; empirical study; data mining.



## LIST OF FIGURES

Figure 1 – Commit c90048ca, commons-beanutils project, available in Github. . . . .	14
Figure 2 – Commit 9cc0604, commons-compress project, available in Github. . . . .	15
Figure 3 – Research Workflow . . . . .	17
Figure 4 – Issue BEANUTILS-157, commons-beanutils project, available in Jira. . . . .	35
Figure 5 – Commit 3a4fa46, commons-beanutils project, available in Github. . . . .	35
Figure 6 – Research Workflow . . . . .	39
Figure 7 – Number of Atoms of Confusion versus Reported Issues by Type . . . . .	40
Figure 8 – Distribution of Commits with AC Removals per Project . . . . .	42
Figure 9 – Projects with more than 8 commits removing ACs . . . . .	43
Figure 10 – Ratio of Commits with At Least One AC Removal . . . . .	44
Figure 11 – Bug-fix x Non-Bug-fix Commits . . . . .	45
Figure 12 – Commit acb09a57, commons-pool project, available in Github . . . . .	47
Figure 13 – Commit Message acb09a57 , commons-pool project, available in Github . .	48
Figure 14 – Commit cf39e4cd, commons-bcel project, available in Github . . . . .	48
Figure 15 – Issue BCEL-197 , commons-bcel project, available in Jira . . . . .	49
Figure 16 – Bug-fix and Improvement x All Other Commits . . . . .	51

## LIST OF TABLES

Table 1 – Apache Software Foundation’s Issues Classification . . . . .	17
Table 2 – Atoms of Confusion Detected by BOHR from MENDES <i>et al.</i> . . . . .	27
Table 3 – Summary of the Main Background and Related Work . . . . .	28
Table 4 – Subject Projects . . . . .	33
Table 5 – Projects Dataset . . . . .	36
Table 6 – Commits With ACs Removals . . . . .	41
Table 7 – Removed ACs per Commit Type . . . . .	44
Table 8 – Atoms of confusion that are likely to be the trigger of a maintenance tasks . .	46
Table 9 – Jira Dataset . . . . .	65
Table 10 – Issue Developer Comments Dataset . . . . .	66
Table 11 – Issue Changelog Dataset . . . . .	66
Table 12 – Issues-Commits Dataset . . . . .	66
Table 13 – Commit Log Dataset . . . . .	67
Table 14 – Git Dataset . . . . .	68
Table 15 – BOHR Detailed Dataset . . . . .	71
Table 16 – BOHR Aggregated Dataset . . . . .	71
Table 17 – Changed/Deleted ACs Dataset . . . . .	72
Table 18 – Commit ACs Changes Dataset . . . . .	73

## **LIST OF ABBREVIATIONS AND ACRONYMS**

AC	Atom of Confusion
ACs	Atoms of Confusion
ASF	Apache Software Foundation
ITS	Issue Tracking Systems
OSS	Open Source Software
VCS	Version Control Systems

## CONTENTS

1	<b>INTRODUCTION</b> . . . . .	12
1.1	<b>Context</b> . . . . .	12
1.2	<b>Motivation</b> . . . . .	13
1.3	<b>Research Questions</b> . . . . .	14
1.4	<b>Study Methodology</b> . . . . .	16
1.5	<b>Goals and Contributions</b> . . . . .	19
1.6	<b>Document Organization</b> . . . . .	20
2	<b>BACKGROUND AND RELATED WORK</b> . . . . .	21
2.1	<b>Program Comprehension</b> . . . . .	21
2.2	<b>Atoms of Confusion</b> . . . . .	22
2.3	<b>Conclusion</b> . . . . .	27
3	<b>DATASET</b> . . . . .	30
3.1	<b>Introduction</b> . . . . .	30
3.2	<b>Dataset Creation Process</b> . . . . .	31
3.2.1	<i>Dataset Preliminaries</i> . . . . .	31
3.2.2	<i>Selection of Java Projects</i> . . . . .	32
3.2.3	<i>Tools Used To Collect The Data</i> . . . . .	32
3.2.3.1	<i>Python Jira</i> . . . . .	32
3.2.3.2	<i>PyDriller</i> . . . . .	33
3.2.3.3	<i>BOHR</i> . . . . .	33
3.2.4	<i>Data Collection Process</i> . . . . .	34
3.3	<b>Dataset Description</b> . . . . .	36
3.4	<b>Conclusion</b> . . . . .	37
4	<b>RESULTS AND DISCUSSION</b> . . . . .	38
4.1	<b>Preliminary Concepts</b> . . . . .	38
4.2	<b>RQ1. To what extent do atoms of confusion relate to the type of maintenance tasks?</b> . . . . .	39
4.3	<b>RQ2. In what type of maintenance tasks are atoms of confusion more frequently removed?</b> . . . . .	40
4.4	<b>RQ3. How often are atoms of confusion likely to directly contribute to a maintenance task?</b> . . . . .	44

4.5	<b>Results Discussion</b> . . . . .	49
4.6	<b>Implications for Researchers</b> . . . . .	51
4.7	<b>Implications for Practitioners</b> . . . . .	52
4.8	<b>Conclusion</b> . . . . .	53
5	<b>CONCLUSION</b> . . . . .	55
5.1	<b>Final Considerations</b> . . . . .	55
5.2	<b>Main Contributions</b> . . . . .	55
5.3	<b>Threats to Validity</b> . . . . .	56
5.3.1	<i>Conclusion Validity</i> . . . . .	56
5.3.2	<i>Internal Validity</i> . . . . .	57
5.3.3	<i>Construct Validity</i> . . . . .	57
5.3.4	<i>External Validity</i> . . . . .	58
5.4	<b>Future Work</b> . . . . .	58
	<b>BIBLIOGRAPHY</b> . . . . .	60
	<b>APPENDIX A –DATASET DESCRIPTION</b> . . . . .	64

## 1 INTRODUCTION

This chapter provides a general overview of the research. Sections 1.1 and 1.2 offer insights into the context and motivation behind the study. Following this, Section 1.3 introduces the research questions that guided the investigation. Section 1.4 details the applied methodology. Moving forward, Section 1.5 briefly summarizes the key objectives and contributions of this dissertation. Finally, Section 1.6 explains the organization of this document, presenting a roadmap for readers to navigate the forthcoming chapters.

### 1.1 Context

Software development is a complex activity that requires technical knowledge and great abstraction skills (BANKER *et al.*, 1998). Although the final product of this activity is code, the development process is much broader and involves several other tasks besides writing code (WEINBERG, 1971). Computer software could be so large and complex that developers must create a mental model associated with the program's operation and establish relationships between it and the source code. The construction of this model is the basis for program comprehension (SINGER *et al.*, 2010; ROBILLARD *et al.*, 2004; FREY *et al.*, 2011).

Program comprehension is the process of understanding how a software system works, particularly emphasizing its source code (BENNETT *et al.*, 2002). It is an essential activity for the entire software development lifecycle. Software engineers must spend a substantial amount of time exploring source code and other artifacts (e.g., documentation and test files) to identify and comprehend the subset of the code relevant to any intended change (SINGER *et al.*, 2010). In fact, previous studies have shown that more than half of the total time spent on software development activities is used for code comprehension (MINELLI *et al.*, 2015; XIA *et al.*, 2017).

The methods employed to comprehend software may vary among developers based on their personality, experience, skills, tasks, and technology used. Knowing the fundamentals of program comprehension is vital for software developers to guarantee the maintenance and evolution of complex software systems (MAALEJ *et al.*, 2014). While performing program comprehension, developers may misjudge the software's actual behavior. Small, self-contained, and indivisible code patterns can cause this confusion. Gopstein *et al.* (2017) named those code snippets in the C programming language as Atoms of Confusion (ACs). They can cause

difficulties during program understanding, which may negatively impact the productivity of a software development team. Also, according to the authors, ACs could introduce defects into the system and increase costs (GOPSTEIN *et al.*, 2017).

Since Gopstein *et al.* (2017)'s discoveries, researchers have been investigating the presence, prevalence, and effects of ACs in various systems built in languages such as C and C++ (GOPSTEIN *et al.*, 2018), Java (LANGHOUT; ANICHE, 2021), and JavaScript (TORRES *et al.*, 2023). For instance, Gopstein *et al.* (2018) used 14 popular C and C++ open-source projects to evaluate the significance and prevalence of ACs in real-world systems. They showed a strong correlation between ACs and bugs in the analyzed projects. Langhout e Aniche (2021) replicated the first Gopstein *et al.* (2017) 's methodology and defined 14 ACs in Java (LANGHOUT; ANICHE, 2021). Inspired by these researches, Mendes *et al.* (2021), Mendes *et al.* (2022) developed a tool to find ACs in Java source code to evaluate the prevalence of ACs in the Java ecosystem, finding more than ten thousand occurrences of ACs in long-lived Java projects (MENDES *et al.*, 2021; MENDES *et al.*, 2022).

## 1.2 Motivation

Previous studies on ACs in Java-based systems primarily focused on their definition, the demonstration of confusion induced by these code snippets, and their prevalence in real-world systems. However, there remains a dearth of studies aimed at comprehensively understanding the implications of using such code patterns in the software development lifecycle, including their potential impact on bug occurrences and subsequent refactoring. To address this research gap and evolve the previous research in Mendes *et al.* (2022), we decided to conduct a study to investigate the **impact of ACs** on Java-based systems. We compiled a comprehensive dataset that links more than 8,000 commits, 4,000 reported issues, and 7,000 ACs from 21 open-source Java projects.

While collecting and analyzing the data from our research, we discovered compelling examples supporting a positive correlation between the number of ACs and the number of reported bugs and improvements. For instance, the bug-fix commit hash c90048ca<sup>1</sup> (see Fig. 1) in the commons-beanutils project modifies a single line of code. This line contains the Atom of Confusion (AC) named Infix Operator Precedence, which occurs when more than one type of binary operator is used in the same code instruction. Confusion regarding this AC arises from

<sup>1</sup> <https://github.com/apache/commons-beanutils/commit/c90048ca>

developers not understanding the execution order of these operators, such that the original code failed to use the order that would satisfy developer needs, resulting in a bug that had to be fixed later by adding brackets to adjust the order of operators. Another example is the improvement commit hash 9cc0604<sup>2</sup> (see Fig. 2) in the commons-compress project, which directly rewrites the ternary operation, which abbreviates the if-then-else structure. The ternary operator may cause code confusion, classified as Conditional Operator atom of confusion. Fig. 2 shows the refactoring performed in the commit that only changes the ternary operation in the source code, which may indicate that the improvement was just to ensure better readability.

```

@@ -218,7 +218,7 @@ protected Object parse(Object value, String pattern) throws ParseException
 {
 218 218     final Number parsed = (Number) super.parse(value, pattern);
 219 219     double doubleValue = parsed.doubleValue();
 220 220     double posDouble = (doubleValue >= (double)0) ? doubleValue : (doubleValue * (double)-1);
 221 -     if (posDouble != 0 && posDouble < Float.MIN_VALUE || posDouble > Float.MAX_VALUE) {
 221 +     if (posDouble != 0 && (posDouble < Float.MIN_VALUE || posDouble > Float.MAX_VALUE)) {
 222 222         throw new ConversionException("Supplied number is not of type Float: "+parsed);
 223 223     }
 224 224     return new Float(parsed.floatValue()); // unlike superclass it returns Float type
  
```

Figure 1 – Commit c90048ca, commons-beanutils project, available in Github.

These findings motivated us to carry out a qualitative analysis aiming to find other code confusion induced by ACs that were likely to contribute directly to the occurrence of a bug or improvement. For this purpose, we inspected bug-fix and improvement commits with AC removals to understand whether such ACs might have directly contributed to the code change.

### 1.3 Research Questions

The primary research objective is to investigate the impact of ACs on long-lived Java systems thoroughly. Specifically, we aim to identify if there are significant correlations between the presence of ACs and the subsequent occurrence of bugs. Furthermore, we also examined if there are correlations between ACs and code improvements within these systems.

The study focuses on addressing the following research questions:

**RQ1.** *To what extent do atoms of confusion relate to the type of maintenance tasks?*

<sup>2</sup> <https://github.com/apache/commons-compress/commit/9cc0604>



```

@@ -143,6 +143,8 @@
143 143  */
144 144
145 145  public class TarArchiveEntry implements TarConstants, ArchiveEntry {
146 +   private static final TarArchiveEntry[] EMPTY_TAR_ARCHIVE_ENTRIES = new TarArchiveEntry[0];
147 +
146 148  /** The entry's name. */
147 149  private String name = "";
148 150
@@ -945,11 +947,14 @@ public boolean isSparse() {
945 947  */
946 948  public TarArchiveEntry[] getDirectoryEntries() {
947 949  if (file == null || !file.isDirectory()) {
948 -   return new TarArchiveEntry[0];
949 +   return EMPTY_TAR_ARCHIVE_ENTRIES;
950 +
949 951  }
950 952
951 953  String[] list = file.list();
952 -   TarArchiveEntry[] result = new TarArchiveEntry[list == null ? 0 : list.length];
953 +   if (list == null) {
954 +     return EMPTY_TAR_ARCHIVE_ENTRIES;
955 +   }
956 +   TarArchiveEntry[] result = new TarArchiveEntry[list.length];
957 +
953 958
954 959  for (int i = 0; i < result.length; ++i) {
955 960  result[i] = new TarArchiveEntry(new File(file, list[i]));

```

Figure 2 – Commit 9cc0604, commons-compress project, available in Github.

To find correlations between bugs and ACs, Gopstein *et al.* (2018) hypothesized that if atoms of confusion cause confusion and are used frequently, their effects should be measurable at the project level. Thus, they analyzed the correlation between the reported bugs for 14 C/C++ projects and the corresponding numbers of ACs, concluding that projects with more ACs tend to have more bugs (GOPSTEIN *et al.*, 2018).

Drawing inspiration from this study, we conducted a similar analysis to investigate and gain insights into this phenomenon within Java projects. Specifically, we explored the correlation between the number of ACs and distinct reported issue types across 21 Java projects, which were carefully selected based on the criteria explained in Section 3.2.2.

**RQ2.** *In what type of maintenance tasks are atoms of confusion more frequently removed?*

Directly measuring the cause of bugs is challenging, but one of the most effective proxies is the code that is changed in a bug-fix commit. Many software projects maintain repositories where they document reports of incorrect behavior in their code and keep track of how and when each issue is resolved. By analyzing the version control histories for each bug reported, we can determine which code was modified to fix the problem. Code removed

in a bug-fix commit is more likely to have contributed to the bug than the code removed in a non-bug-fix commit. Previous studies showed that ACs are 1.25x as likely to be removed in a bug-fix commit than a non-bug-fix commit in GCC (GOPSTEIN *et al.*, 2018).

In this direction, to observe such events in Java and consider more projects, we utilized the issue classification already provided by project developers in Jira to classify the commits from 21 open-source projects into bug, improvement, task, sub-task, new feature, wish, and test, according to the Apache Software Foundation’s wiki<sup>3</sup>, as shown in Table 1. This allows us to compare the rate at which ACs are removed in bug-fixing versus the other types of commits. This comparison can provide a proxy for determining whether atoms of confusion are more likely to be responsible for bugs (GOPSTEIN *et al.*, 2018).

**RQ3.** *How often are atoms of confusion likely to directly contribute to a maintenance task?*

To address RQ2, we investigated in what kind of issues ACs were more frequently removed. However, the fact that ACs disappear may not necessarily mean that they were intentionally removed. They may disappear simply because developers removed the piece of code that contained them. So, to improve deeper insights and enhance our understanding, a qualitative analysis was conducted to assess whether the removed AC could have directly contributed to a bug-fix or improvement task.

## 1.4 Study Methodology

Fig. 3 shows the research workflow. **First**, we selected 21 open-source Java projects as the object of our empirical study, according to the criteria discussed in Section 3.2.2.

In the **second step**, our objective was to establish a connection between the reported issues and the respective commits that resolve them. To accomplish this, we developed a Python script that extracted and processed information from both the project management tool Jira<sup>4</sup> and the Git version control system<sup>5</sup>, tools used by all 21 projects under study. All project data was mined from their inception until May 22, 2023, the date when the dataset was built.

At this stage, we used the issue classification already provided by project developers to classify commits. According to the Apache Software Foundation’s wiki, issues are classified following the categories: bug-fix, improvement, new feature, sub-task, task, test, or wish. Table

<sup>3</sup> <https://cwiki.apache.org/confluence/display/FLUME/Classification+of+JIRA+Issues>

<sup>4</sup> <https://issues.apache.org/jira>

<sup>5</sup> <https://github.com/apache>

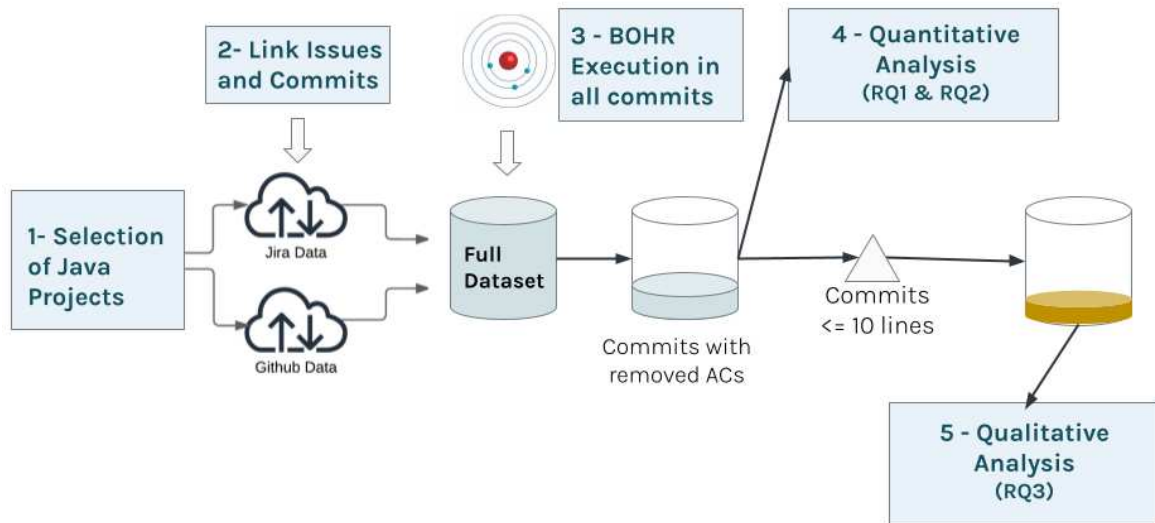


Figure 3 – Research Workflow

1 shows their definition.

Hence, 8,641 commits were classified based on the types of their 4,862 associated issues in Jira. To facilitate this process, we utilized the Python data mining frameworks Pydriller<sup>6</sup> and Jira-Python<sup>7</sup> (SPADINI *et al.*, 2018; JIRA-PYTHON-LIBRARY, 2012).

Table 1 – Apache Software Foundation’s Issues Classification

bug	It’s a defect in the source code.
improvement	It’s an improvement or enhancement to an existing feature.
new feature	It’s a new feature that hasn’t been developed yet.
task	It’s a task that needs to be done that doesn’t fall under any of the other issue types.
sub-task	It’s a child of a task
test	It’s a new unit or integration test.
wish	It’s wish-list items that could be classified as new feature or improvement

To establish the connection between a commit and a Jira issue, we searched the commit messages to identify the corresponding issue ID. The Apache Software Foundation (ASF) projects follow this strategy to ensure traceability between source code modifications and project management activities (VIEIRA *et al.*, 2019).

After that, we used the BOHR tool (**third step**), created by Mendes *et al.* (2022), to count the number of ACs in the latest release of each project. This tool can detect 10 out of the

<sup>6</sup> <https://pydriller.readthedocs.io>

<sup>7</sup> <https://jira.readthedocs.io>

14 types of ACs defined for Java by Langhout e Aniche (2021), as can be observed in Table 2. In this phase, our focus was to answer the first research question detailed in Section 1.3. For that, we used the information regarding the number of issues reported by type and the number of ACs throughout the last release at the time the dataset was built of the 21 subject projects, as specified in Table 4.

Subsequently, we identified the source code of the immediately previous version to each commit. Our goal was to run the BOHR on the previous versions of the classified commits to find the atoms of confusion existing in this version and search for them in the lines of code removed from each classified commit. This step was necessary because the BOHR tool uses Spoon's open-source code library, which needs to have the structured source code of a project or class as input. Hence, it is not possible to apply the BOHR tool to independent pieces of source code. For this reason, we did not use it directly on the added and removed lines of a commit (MENDES *et al.*, 2021; MENDES *et al.*, 2022; PAWLAK *et al.*, 2015). Therefore, we executed BOHR for all 8,641 analyzed commits to count the total number of ACs present in previous versions and those after the commit, generating reports on the prevalence of ACs.

In the **fourth step**, Python scripts were built to search for the atoms identified in the version immediately preceding each commit, analyzing their removed and added lines, thus determining the ACs that were removed or changed in each commit, answering the second research question described in Section 1.3.

In the **fifth step**, we focused on selecting a subset of bug-fix and improvement commits that involved the removal of ACs. This step aimed to gain a deeper understanding and evaluate whether the atom of confusion was likely to be the primary factor behind these maintenance tasks. To facilitate a manual analysis, we only considered ACs removed from commits with a maximum of ten lines deleted. This decision was based on the recognition that as the number of lines deleted in a commit increased, it became more challenging, fuzzy, and time-consuming to determine the specific impact of the AC on the bug or improvement.

We thoroughly examined the commit messages, source code, title, description, and developers' comments related to the corresponding issue during the manual analysis. Our objective was to ascertain if the atom of confusion played a significant role in causing the bug or improvement. For each AC, we documented whether it likely contributed directly to the occurrence of the bug or improvement. Additionally, we specified the evidence that led to our conclusion in the dataset.

## 1.5 Goals and Contributions

There are some studies about Atoms of Confusion including definitions and confusion in code comprehension (GOPSTEIN *et al.*, 2017; CASTOR, 2018), the prevalence and impacts of ACs in C/C++ software projects (GOPSTEIN *et al.*, 2018), ACs in the context of Java programming language (LANGHOUT; ANICHE, 2021; MENDES *et al.*, 2021; MENDES *et al.*, 2022), etc. However, to the best of our knowledge, our study is the first to establish a clear connection between Atoms of Confusion (ACs) and the source of bugs or the trigger of improvements in Java projects.

This dissertation aims to provide a first insight into the impact of ACs in Java projects. In addition to a study of the relationship between ACs and maintenance tasks in Java projects, we also intend to provide a dataset with information on three perspectives: project management, code versioning, and atoms of confusion from 21 open-source Java projects. Furthermore, all the methodology and source code required to create the dataset are available at (PINHEIRO, 2023) so that other researchers can expand it.

During our analysis, we detected a positive correlation between the number of ACs and the number of reported bugs and improvements on the studied Java projects. We also explored changes in commits, aiming to gain a better understanding of the context in which ACs are removed. Since each commit is associated with at least one reported issue (such as a bug or improvement), we were able to compare the ratio of AC removal across different types of commits. This served as a proxy to assess whether ACs are likely contributing to reported issues. Our findings revealed a higher ratio of removed ACs in bug-fix and improvement commits compared to other types of commits (such as task, sub-task, new feature, wish, and test) in 14 out of the 19 studied projects where ACs were removed in commits.

In conclusion, to complement our quantitative findings, we conducted a qualitative analysis to gain a deeper understanding of how frequently atoms of confusion played a role in causing bugs or improvements. We examined ACs that were removed in commits related to bugs or improvements, specifically those with up to ten lines removed. Our analysis involved scrutinizing the source code, commit messages, as well as the title, description, and comments of associated Jira issues. Out of a total of 8,641 commits across 21 analyzed projects, 391 involved the removal of ACs. Among them, 53 met the criteria for our qualitative analysis. In 7 of these commits, 9 removed ACs were deemed likely to have directly contributed to the occurrence of a bug or improvement. To the best of our knowledge, our research is the first to explore the

connection between Atoms of Confusion and the origin of bugs or the reasons for improvements in Java projects.

We believe our findings motivate researchers and developers to investigate the presence of ACs further and propose tools for detecting and refactoring such code fragments, as ACs not only contribute to confusion but may also be the cause of system bugs and the trigger of improvements.

## **1.6 Document Organization**

The remainder of this work is structured as follows. Chapter 2 discusses the background and related work. Chapter 3 presents the dataset we built to support this study. The results and discussion are described in Chapter 4. Finally, in Chapter 5, we provide the final considerations, threats to validity, and proposals for further investigation.

## 2 BACKGROUND AND RELATED WORK

This chapter presents the main concepts, definitions, and related works associated with this research. This chapter is organized into three sections. Section 2.1 discusses the topic of Program Comprehension, while Section 2.2 presents the works about Atoms of Confusion. These sections explore the key concepts, motivations, objectives, state of the art, and related works regarding this study. Finally, Section 2.3 provides the concluding remarks of this chapter.

### 2.1 Program Comprehension

Software development is a complex activity that requires technical knowledge and great abstraction skills (BANKER *et al.*, 1998). Although the final product of this activity is code, the software development process is much broader and involves various other tasks besides writing (WEINBERG, 1971). Software systems are so vast and intricate that developers need to establish connections between the source code and the associated mental model that the programmer must create. The construction of this model forms the foundation for Program Comprehension (SINGER *et al.*, 2010; ROBILLARD *et al.*, 2004; FREY *et al.*, 2011).

Program comprehension is a cognitive process in which developers consume and produce a significant amount of knowledge about software (MAALEJ *et al.*, 2014). To perform this activity, software engineers seek to understand how a system works, with its source code as the primary reference. Program comprehension requires an understanding and study of the user's domain of the system, software engineering, and technical programming knowledge (BENNETT *et al.*, 2002). Areas such as documentation, visualization, design, analysis, refactoring, and reengineering, among others, are driven by the need for program comprehension (RAJLICH; WILDE, 2002).

Thus, program comprehension becomes an essential part of every stage in the software development process, especially during the phases of evolution and maintenance. After all, software that is not understood cannot be modified. According to Singer *et al.* (2010), program comprehension primarily takes place before changes in the software system, as developers need to explore the source code and other artifacts to identify and understand the subset of code relevant to their objectives. The strategies employed in this activity vary among professionals and depend on their personalities, experiences, and skills, as well as the types of tasks they need to perform and the related technologies (SINGER *et al.*, 2010).

Previous studies indicate that developers spend up to 50% of their working time searching for information to answer questions about the system under development (MURPHY *et al.*, 2006; KO *et al.*, 2007). In this vein, Minelli *et al.* (2015) conducted an experiment to assess how developers allocate their time, concluding that, on average, program comprehension requires 70% of the development time (MINELLI *et al.*, 2015). Similarly, Fjeldstad (1983) reported that during software maintenance, programmers use approximately half of their working time understanding code (FJELDSTAD, 1983). Corroborating these findings, Xia *et al.* (2017) found in their study that, on average, developers dedicate 58% of their time to code comprehension activities (XIA *et al.*, 2017).

Therefore, this activity is a crucial aspect of both software development and maintenance, as programmers dedicate a significant portion of their time to code comprehension. The more effort required for this task, the less time is available for other development-related activities, such as code modification and navigation (RAHMAN, 2018). Thus, understanding how a program works is essential for software engineers, and given its importance, this topic is the focus of various studies and experiments aimed at evaluating approaches and techniques that seek to enhance program comprehension (SCHRÖTER *et al.*, 2017).

Although research in this field has evolved considerably in recent years, little is still known about how developers practice program comprehension in their daily work. For this reason, Maalej *et al.* (2014) conducted a quantitative and qualitative study to understand the strategies, tools, and knowledge applied by developers when performing the activity of code comprehension in practice. As a result, it was found that there is a gap between research and practice since no use of program comprehension tools was observed, and developers seem to be unaware of them (MAALEJ *et al.*, 2014).

Therefore, exploring this subject is highly relevant to the literature, as enabling a better understanding of the source code of a system assists various professionals, such as developers, system analysts, software architects, testers, among others, in their daily practical work, contributing to higher-quality software development and maintenance.

## **2.2 Atoms of Confusion**

Related to the topic of program comprehension, humans often misinterpret the meaning of source code, which can lead to an improper assessment of the actual behavior of a software system. In this context, confusion can be defined when programmers and the computer



arrive at different conclusions about the behavior of the same piece of code. Such confusion can be caused by small and isolated code patterns that Gopstein *et al.* (2017) referred to as Atoms of Confusion (ACs). An atom of confusion is defined as the smallest, indivisible portion of source code that can confuse developers, potentially leading them to a misjudgment of code comprehension, which can result in bugs and, consequently, have impacts such as decreased productivity and increased costs in software development (GOPSTEIN *et al.*, 2017).

Gopstein *et al.* (2017) empirically demonstrated, in a controlled experiment with Computer Science students experienced in the C programming language, that certain code patterns can lead to a significant increase in misinterpretations by developers when compared to functionally equivalent code without such patterns. This study identified 15 statistically significant atoms of confusion in the C language. Subsequently, a second experiment was conducted to evaluate the impact, in terms of confusion intensity, caused by these atoms on the experiment participants. Thus, this work defined a methodology for empirically deriving ACs, a large, publicly available dataset for other researchers to replicate and extend the experiments, and a survey of well-known and popular C style guidelines that recommend the use of ACs, which contradicts the findings of this study (GOPSTEIN *et al.*, 2017).

To delve deeper into the topic, Castor (2018) presented a more detailed definition for an atom of confusion as a code pattern with the following characteristics:

- Precisely identifiable;
- Likely to cause confusion;
- Replaceable by a functionally equivalent snippet that causes less confusion;
- Indivisible.

In their work, previous studies related to ACs were applied to the Swift programming language, resulting in a set of 6 candidate atoms of confusion in this language (CASTOR, 2018).

Intending to detect and compare the brain activity of developers while analyzing functionally equivalent code snippets with and without confusion code, Yeh *et al.* (2017) employed an EEG (electroencephalogram) device to assess possible differences that might indicate increased cognitive effort when trying to comprehend code with confusion. The results were promising, as they indicated that more neurons were activated or oscillated in harmony while participants analyzed confusing code snippets. Furthermore, the work demonstrated that experiments involving a deeper analysis of brain activity are feasible and promising (YEH *et al.*, 2017).

In a similar vein, Lewis *et al.* (2018) showed in their EEG studies that atoms of confusion caused significant confusion among experiment participants (LEWIS *et al.*, 2018). To detect the visual attention of programmers while comprehending source code, Oliveira *et al.* (2020) conducted research using an eye tracker to analyze the distribution of visual attention during the evaluation of code with and without ACs. From an aggregate perspective, an increase of 43.02% in time and 36.80% in gaze transition was observed when experiment participants evaluated code with ACs. It was also confirmed that regions with ACs received the highest attention from the eyes. These results corroborate the fact that atoms of confusion impact comprehension and, consequently, the performance of developers (OLIVEIRA *et al.*, 2020).

Until then, there had been no assessment of the impact of atoms of confusion on real-world systems. Therefore, Gopstein *et al.* (2018) studied a set of 14 of the most popular and important open-source projects in C and C++ to measure the prevalence and relevance of the 15 atoms of confusion defined in a previous work. The results showed that they frequently occur in all studied projects, such as the Linux kernel and GCC, appearing on average once every 23 lines of code. Additionally, it was found that there is a strong correlation between the presence of ACs and subsequent bug-fix commits. It was also inferred that code snippets containing ACs are more likely to have comments in the code. Another explored relationship was at the project level, indicating that the rate of security vulnerabilities is higher in projects with more ACs. This demonstrates that atoms of confusion are prevalent, occurring frequently in real projects, and significant, being removed by bug-fix commits at a high rate (GOPSTEIN *et al.*, 2018).

In a similar vein of estimating the impacts of confusing code patterns on real-world projects, Medeiros *et al.* (2019) conducted repository mining, analyzing 50 open-source projects in C, including Apache, Redis, OpenSSL, and Python. This allowed them to find over 109,000 occurrences of confusing code patterns in these projects, showing that 92% of these code patterns are indeed used in practice by developers of popular and relevant systems. Additionally, a survey was conducted with developers of open-source projects to assess their perception of confusing code patterns, with the majority agreeing that 50% of the analyzed patterns do indeed cause confusion. In this work, the project guidelines were also analyzed to understand the guidelines provided by these projects to developers regarding confusing code patterns. It was discovered that only a few of these projects address the issue in their guidelines. Lastly, to measure the importance given to the topic by developers of these projects, 35 random pull requests were made to open-source projects, replacing code with a confusing pattern with a functionally equivalent

but non-confusing snippet. As a result, only 8 out of 35 pull requests were accepted, 14 received no feedback, and 13 were rejected, with responses indicating that the code was working without errors and the current version was adequate. This demonstrates that the topic still does not receive the attention it deserves from many developers (MEDEIROS *et al.*, 2019).

In further assessing the impacts of code confusion, Ebert *et al.* (2019) studied this topic in the context of code review, which is a widely used software quality assurance practice with potential benefits such as defect detection, knowledge transfer, and adherence to project code standards. The study showed that confusion during the code review process can delay merge decisions, increase the need for discussion about a specific piece of code, and lower the quality of the review. Moreover, the results indicated that confusion can lead developers to approve a code change even without fully understanding it, which poses a risk to the related system (EBERT *et al.*, 2019).

On the other hand, in a recent study within the context of code reviews, Bogachenkova *et al.* (2022) investigated the possibility of a relationship between the presence of ACs and confusion in code reviews. Using a tool to detect atoms of confusion and manual analysis of comments in code reviews, the statistical analysis performed did not reveal any relationship between ACs and confusion in code reviews. Additionally, the results showed that ACs present in pull requests are not eliminated after the review and acceptance of the pull request (BOGACHENKOVA *et al.*, 2022).

In more recent work, Gopstein *et al.* (2020) conducted a qualitative investigation related to code comprehension and atoms of confusion to provide context for the results of previous studies. They explored research gaps in the area of ACs, focusing on how and why ACs confuse developers, while prior work had mainly focused on defining and quantifying ACs without delving into these aspects. The study aimed to understand and describe how programmers assess a piece of code, the step-by-step process they follow, the difficulties they encounter, and how they address them. The study revealed that not all misinterpretations originated from the same source, that some correct answers were obtained through incorrect reasoning, and that confusion could exist even when arriving at a correct answer. Consequently, it was concluded that research might be underestimating the number of misinterpretations caused by ACs because in experiments, getting the output of a particular piece of code correct does not necessarily mean that the reasoning was correct, or that it was easy, or that it was not confusing to reach that result. Thus, the importance of considering a more flexible model that assesses multiple aspects, not just

whether the output of a code snippet is correct or not, was observed (GOPSTEIN *et al.*, 2020).

Building on the studies and methodology defined by Gopstein *et al.* (2017), Langhout e Aniche (2021) generalized and replicated the knowledge about atoms of confusion for the Java programming language. In this work, 14 ACs were defined in Java. Afterward, the impact of these atoms was assessed through an experiment involving Computer Science students. The results showed that participants were 2.7 to 56 times more likely to misinterpret code snippets containing 7 of the 14 ACs defined for Java. Additionally, students reported that code snippets with 10 of the 14 ACs were more confusing and/or less readable than their functionally equivalent code without the respective ACs (LANGHOUT; ANICHE, 2021).

In the context of ACs in Java code, Mendes *et al.* (2021) introduced a tool called BOHR - The Atoms of Confusion Hunter, designed to assist in the automated identification of ACs in Java code. It produces reports on the prevalence of 8 out of the 13 ACs types pointed out by Langhout e Aniche (2021). Furthermore, BOHR provides an API to expand the search for existing ACs and create search mechanisms for new ACs. The tool's accuracy was validated through the analysis of three open-source Java projects, demonstrating 100% accuracy in correctly identifying code snippets containing ACs, their types, the involved class names, and the line numbers of their occurrences. In an evolution of this work, Mendes *et al.* (2022) demonstrated that the tool now detects two additional AC types identified by Langhout e Aniche (2021). To assess the precision and recall of the tool, a dataset was constructed and made available, based on four open-source Java projects, with manual identification of ACs, their respective types, corresponding code snippets, and locations. After discussions and adjustments, BOHR achieved 100% precision and identified all the ACs in the dataset. Moreover, with the support of this tool, Mendes *et al.* (2022) conducted an assessment of the prevalence, co-occurrence, and evolution of ACs in 27 open-source Java libraries. In this study, 11,404 AC occurrences were identified. The AC types with the highest prevalence were the Conditional Operator and Logic as Control Flow. It was also shown that these two ACs were more likely to co-occur in the same class. Lastly, the analysis of the evolution of ACs over the library lifecycle demonstrated that AC occurrences do not decrease; on the contrary, in 13 of the studied libraries, the occurrence of ACs grew proportionally more than the project size in lines of code. Additionally, in 15 libraries, the number of Java classes containing at least one atom of confusion increased over time (MENDES *et al.*, 2021; MENDES *et al.*, 2022).

The BOHR tool can detect 10 out of the 14 types of ACs defined for Java by Langhout

Table 2 – Atoms of Confusion Detected by BOHR from MENDES *et al.*

Atom of Confusion Type	Snippet with Atom of Confusion	Snippet without Atom of Confusion
Infix Operator Precedence	<code>int a = 2 + 4 * 2;</code>	<code>int a = 2 + (4 * 2);</code>
Post-Increment/Decrement	<code>a = b++;</code>	<code>a = b; b += 1;</code>
Pre-Increment/Decrement	<code>a = ++b;</code>	<code>b += 1; a = b;</code>
Conditional Operator	<code>b = a == 3 ? 2 : 1;</code>	<code>if(a == 3){b = 2;} else{b = 1;}</code>
Arithmetic as Logic	<code>(a - 3) * (b - 4) != 0</code>	<code>a != 3 &amp;&amp; b != 4</code>
Logic as Control Flow	<code>a == ++a &gt; 0    ++b &gt; 0</code>	<code>if(!(a + 1 &gt; 0)) {b += 1;} a += 1</code>
Change of Literal Encoding	<code>a = 013;</code>	<code>a = Integer.parseInt("13", 8);</code>
Omitted Curly Braces	<code>if(a) f1(); f2();</code>	<code>if(a){ f1(); } f2();</code>
Type Conversion	<code>a = (int) 1.99f;</code>	<code>a = (int) Math.floor(1.99f);</code>
Repurposed Variables	<code>int a[] = new int[5]; a[4] = 3; while (a[4] &gt; 0) {   a[3 - v1[4]] = a[4];   a[4] = v1[4] - 1;} System.out.println(a[1]);</code>	<code>int a[] = new int[5]; int b = 5; while (b &gt; 0) {   a[3 - a[4]] = a[4];   b = b - 1;} System.out.println(a[1]);</code>

e Aniche (2021), as can be observed in Table 2.

## 2.3 Conclusion

In this chapter, we discussed the main concepts related to Program Comprehension and Atoms of Confusion, covering definitions, related studies, and the impacts of this topic on the software development process. Table 3 presents a summary of the main background and related work shown in this chapter.

Most of the studies conducted on Atoms of Confusion have focused on demonstrating the confusion caused by these code snippets, as well as their prevalence in real-world systems. However, there are still few studies that aim to better understand the impacts of using these code patterns throughout the software lifecycle, relating them to the occurrence of bugs, the need for comments to better explain such code snippets, and subsequent refactoring, for example.

In this direction, Gopstein *et al.* (2018) studied these impacts in C/C++ projects, concluding that atoms of confusion are 1.25 times more likely to be removed in bug-fix commits than in other commits. Furthermore, it was observed that projects with more ACs also had more bugs and vulnerabilities. The study also found that ACs are 1.13 times more likely to be commented on than code snippets without ACs (GOPSTEIN *et al.*, 2018).

Although Gopstein *et al.* (2018) addressed this research line, no study focusing on the impacts of atoms of confusion was found for Java projects. Therefore, our study aims

Table 3 – Summary of the Main Background and Related Work

Research	Programming Language	Method	Main Findings
Gopstein <i>et al.</i> (2017)	C/C++	Manual code analysis	Definition of the concept of Atom of Confusion and proposition of its 15 types
Gopstein <i>et al.</i> (2018)	C/C++	Prevalence analysis through automatic code analysis	ACs are prevalent, occurring frequently in real projects, and significant, being removed by bug-fix commits at a high rate
Castor (2018)	Swift	Manual code analysis	Detailed definition of AC and 6 ACs for Swift.
Gopstein <i>et al.</i> (2020)	C/C++	Qualitative investigation focusing on how and why ACs cause confusion in developers	Not all misinterpretations originated from the same source, some correct answers were obtained through incorrect reasoning, and confusion could exist even when arriving at a correct answer
Oliveira <i>et al.</i> (2020)	C/C++	Eye-tracking camera to analyze the distribution of visual attention during the evaluation of code with and without ACs	Increase of 43.02% in time and 36.80% in gaze transition was observed when experiment participants evaluated code with ACs. Regions with ACs received the highest attention from the eyes.
Langhout e Aniche (2021)	Java	Translation of ACs from C/C++ to Java and evaluation of the confusion on Java developers	Developers reported that code snippets with 10 of the 14 ACs were more confusing and/or less readable than their functionally equivalent code without the respective ACs
Mendes <i>et al.</i> (2021)	Java	Automated identification of ACs in Java code	Tool BOHR to assist in the automated identification of ACs in Java code. API to expand the search for existing ACs and create search mechanisms for new ACs.
Mendes <i>et al.</i> (2022)	Java	Prevalence analysis through automatic code analysis	ACs are prevalent and the number of occurrences grows over time in Java real systems.

Source: the author.

to contribute to research in Program Comprehension, specifically on Atoms of Confusion, by investigating their impacts on open-source Java projects and their relationship with maintenance tasks. This investigation will be carried out using a Java Atoms of Confusion detection tool, BOHR (MENDES *et al.*, 2021); Git repository mining tools (Pydriller) (SPADINI *et al.*, 2018);

and Jira data analysis tools (Jira-Python) (JIRA-PYTHON-LIBRARY, 2012).

As highlighted, the ability to identify and remove elements that confuse source code goes beyond simply avoiding bugs because understanding a program is one of the most important activities carried out during the software development process, directly impacting productivity, schedules, and project costs.

### 3 DATASET

This chapter explains the motivation, the creation process, and the composition of the dataset produced in this research. Section 3.1 provides the context and motivation for the dataset. Following that, the dataset creation process is detailed in Section 3.2. Next, the dataset is described and detailed in Section 3.3. Finally, Section 3.4 brings the final considerations about this chapter.

#### 3.1 Introduction

Open Source Software (OSS) is the product of collaborative efforts by geographically and temporally dispersed contributors, including professional software developers and volunteers from diverse backgrounds. Despite participating in a highly decentralized process, they manage to work together efficiently and productively. To assist them in this process, Issue Tracking Systems (ITS) and Version Control Systems (VCS) are essential (GERMAN, 2003; CROWSTON *et al.*, 2008).

Issue Tracking Systems (ITS) provide a valuable source of information related to software development. Within these systems, an issue can describe various aspects of software development, including bugs, new functionalities, security vulnerabilities, enhancements to existing functionalities, or project tasks (AL-ZUBAIDI *et al.*, 2017). In OSS projects, ITS tools have a significant role, serving as task management tools and communication channels for stakeholders involved in the issues' life cycle. Thus, researchers and practitioners have been using ITS information to investigate and explore important aspects of the software development process (VIEIRA *et al.*, 2019).

The source code, along with its historical evolution, serves as both the end product and a detailed record of the software development process. It stands as an invaluable asset for the examination and enhancement of software development practices. In this context, Version Control Systems (VCS) are an indispensable tool in software development and collaborative work. VCS enables the meticulous tracking of changes made to source code, offering a historical record of who, when, and what changes were made. This feature aids in debugging, auditing, and understanding the evolution of software (SPINELLIS, 2005; ALWIS; SILLITO, 2009; JERMAKOVICS *et al.*, 2011; MOCKUS, 2009).

To investigate and explore the impacts and the relationship of Atoms of Confusion



with project management aspects and source code changes within a project, we constructed a dataset, inspired in Vieira *et al.* (2019), that links Jira issues to their corresponding GitHub commits, which were used as the Issue Tracking System and Version Control System, respectively, in the 21 projects under study. This allowed us to combine information from three perspectives: project management, code versioning, and Atoms of Confusion. Through the analysis of this dataset, we generated important insights about the potential consequences of using ACs by correlating them with source code changes and, consequently, with an issue reported in Jira, as presented in the following chapter.

In addition to the research questions addressed in this study, we have made the complete dataset available to the community, along with all the necessary scripts for its generation. These resources are not only accessible to other researchers interested in testing their hypotheses using an established dataset, but can also be utilized to enrich the dataset with information from other Java projects that utilize Jira and GitHub. This provides an opportunity for collaboration and extended research about atoms of confusion.

## 3.2 Dataset Creation Process

### 3.2.1 Dataset Preliminaries

The Apache Software Foundation (ASF) represents a decentralized open-source community comprised of developers. Established as a U.S.-based non-profit organization to provide support for Apache Software Projects, ASF consistently releases software under the Apache License, ensuring that all offerings are both free and open source. Their official website<sup>1</sup> lists more than 8,400 committers contributing to more than 320 active projects. Our study specifically focused on 21 ASF projects, which were carefully selected based on the criteria explained in Section 3.2.2. Furthermore, ASF projects are well-established, extensively documented, and popular among developers. This widespread adoption indicates a notable level of trustworthiness from the developers' standpoint.

Jira<sup>2</sup> is a proprietary Issue Tracking System developed by the Australian Atlassian Corporation. Atlassian offers free Jira services to several open-source projects. By default, the ASF projects adopt Jira as their ITS tool. Git<sup>3</sup> is a free and open-source distributed Version

---

<sup>1</sup> <https://www.apache.org/>

<sup>2</sup> <https://www.atlassian.com/software/jira>

<sup>3</sup> <https://git-scm.com/>

Control System used by the subject projects.

### 3.2.2 *Selection of Java Projects*

To initiate our research, we decided to select Java projects in which atoms of confusion presence had already been demonstrated, and sufficient information was available about the commits in these projects. The idea was to utilize the commit category and description (e.g., bug-fix) and developers' comments to extract information regarding the impact of ACs on the projects.

As discussed in Chapter 2, Langhout e Aniche (2021) defined atoms of confusion in Java, while Mendes *et al.* (2022) demonstrated the frequent occurrence of such atoms in 27 real-world Java systems. Inspired by these findings in (MENDES *et al.*, 2022), we decided to follow up on their research and explore the impacts of ACs in these 27 projects. However, upon further examination, we found that only 21 projects, as shown in Table 4, within the Apache Commons ecosystem<sup>4</sup> complied with the criteria of using publicly accessible project management and source code version control tools, facilitating traceability between project issues and associated commits. This accessibility allowed us to acquire the necessary information for our study.

As explained by Mendes *et al.* (2022), all of these projects have automated tests and use build automation systems, contributing to their analysis with Spoon, the library used by BOHR to identify ACs. Moreover, these projects are constantly updated, used by thousands of systems, and have an active lifecycle of over ten years. This demonstrates relevance and significance for analyzing the possible impacts of ACs in them (MENDES *et al.*, 2022).

### 3.2.3 *Tools Used To Collect The Data*

#### 3.2.3.1 *Python Jira*

Python Jira<sup>5</sup> is a Python library designed to ease the use of the Jira REST API, which is used to interact with the Jira Server applications remotely. This tool facilitates the extraction of data from Jira issues, such as priority, description, developers' comments, activities, links to other issues, and status, among others (JIRA-PYTHON-LIBRARY, 2012).

---

<sup>4</sup> <https://commons.apache.org>

<sup>5</sup> <https://jira.readthedocs.io/>

Table 4 – Subject Projects

	<b>Version</b>	<b>LoC</b>	<b>Classes</b>	<b>ACs</b>	<b>ACTypes</b>
bcel	6.7.0	29638	392	370	7
beanutils	1.9.4	11644	111	174	5
cli	1.5.0	2151	23	84	3
codec	1.16.0	9505	80	444	7
collections	4.4	28955	326	565	6
compress	1.23.0	45057	399	1168	9
configuration	2.9.0	20214	261	375	6
dbcp	2.9.0	14454	66	127	2
dbutils	1.8	3149	50	32	2
digester	3.2	9917	168	94	5
email	1.5	2815	23	50	5
exec	1.3	1757	32	38	4
fileupload	1.5	1861	34	33	6
functor	1.0	5861	158	495	3
io	2.13.0	17370	244	415	7
lang	3.12.0	29745	215	880	8
math	4.0	71856	721	2630	8
net	3.9.0	17179	213	402	6
pool	2.12.0	5931	52	87	5
proxy	2.0	2435	54	15	2
validator	1.7	7619	64	167	5

### 3.2.3.2 *PyDriller*

PyDriller<sup>6</sup> is a Python framework designed for Git repositories mining, offering a straightforward way to extract data from them. This tool provides easy extraction of various information from a Git repository, including commit messages, developer statistics, modifications, diffs, and the source code associated with a particular commit (SPADINI *et al.*, 2018).

### 3.2.3.3 *BOHR*

The Atoms of Confusion Hunter (BOHR)<sup>7</sup> is a tool designed to assist in the automated identification of ACs in Java code. It generates reports indicating the prevalence of these atoms, the classes in which they occur, the lines of their occurrences, the associated code snippets, and the types of atoms found (MENDES *et al.*, 2021; MENDES *et al.*, 2022).

<sup>6</sup> <https://pydriller.readthedocs.io>

<sup>7</sup> <https://github.com/wendellmf/bohr-aoc-api>

### 3.2.4 Data Collection Process

The dataset was constructed through an automated mining process using the Python 3.10.4 programming language to extract, process, and analyze the data. All data was sourced from the official Apache Software Foundation repositories of Jira<sup>8</sup> and Git<sup>9</sup>.

The process began by extracting information from Jira utilizing the Python Jira library. During this phase, we retrieved all issues from 21 subject projects having ‘CLOSED’ or ‘RESOLVED’ status, a ‘FIXED’ resolution field, and resolved up to May 22, 2023, which is the date when the dataset was generated.

To enable tracking between a reported issue on Jira and the commits performed to resolve it, developers in ASF projects follow the practice of including the Jira issue ID in the commits messages dedicated to addressing the issue (VIEIRA *et al.*, 2019; RATH; MÄDER, 2019). As a result, we employ these issue IDs to trace and extract information about source code changes from commits that fix issues using Pydriller. Figure 4 shows an example of issue BEANUTILS-157 of project BEANUTILS as represented in Jira<sup>10</sup>. The VCS captures the evolution of a project’s source code in the form of changesets according to the example of commit 3a4fa46<sup>11</sup> shown in the Figure 5. It is possible to observe the issue ID in the commit message. This process reliably discovers traceability links among changesets of the VCS and referenced issue artifacts in the ITS. All issues, changesets, and respective properties are part of the dataset.

During this process of linking issues with the commits that resolve them, we observed that there are commits linked to a single issue, while others are associated with multiple issues. Additionally, some commits are not linked to any issue. Furthermore, we noticed that there are issues that are related to more than one commit. As our research requires information at both the source code and project levels, we included in the dataset only those issues and commits that have at least one linkage between them. If a single commit is linked with more than one issue, the commit is considered according to the number of issues it is related to, receiving the classification of each respective issue.

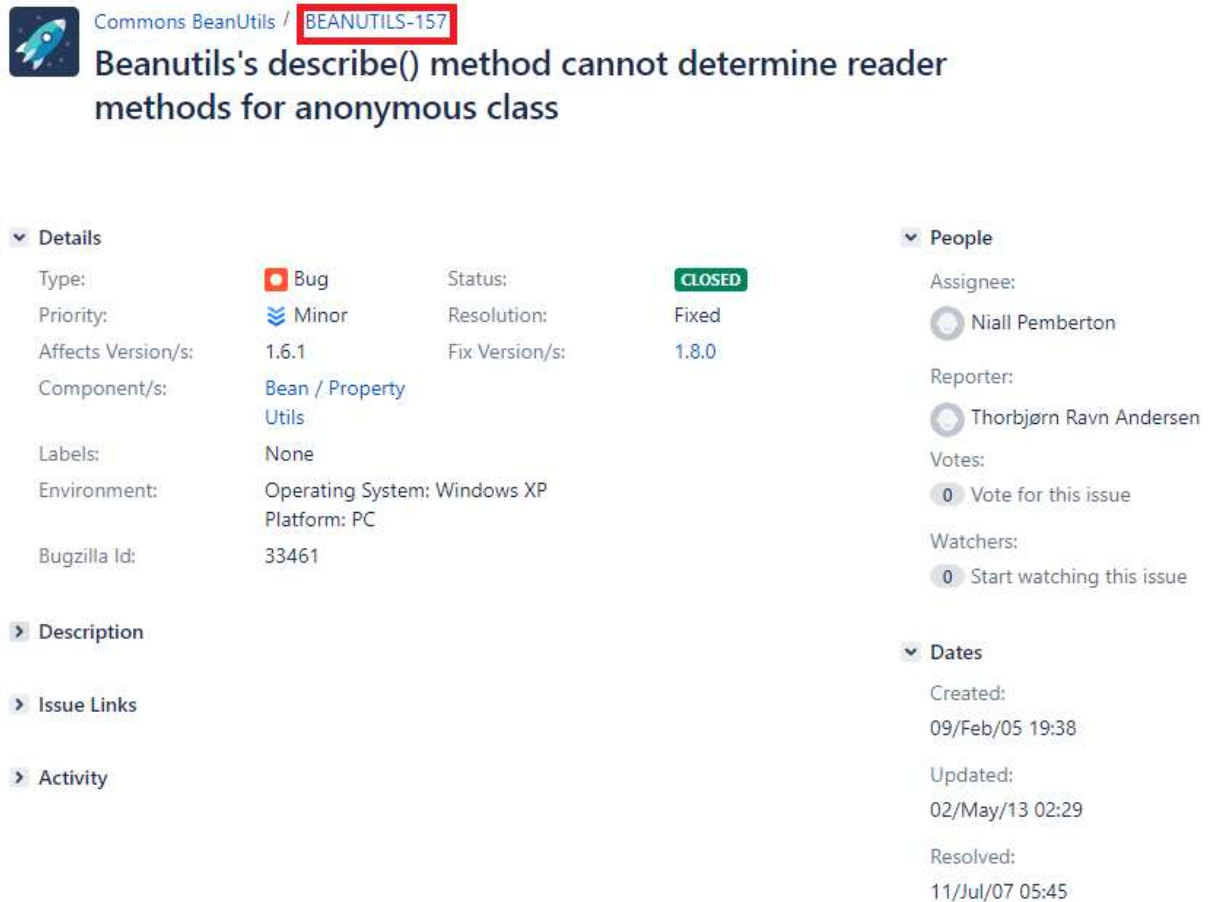
To complement the dataset with information about atoms of confusion, we executed BOHR on each version of the project, represented by each commit mapped in the previous step.

<sup>8</sup> <https://issues.apache.org/jira>

<sup>9</sup> <https://github.com/apache>

<sup>10</sup> <https://issues.apache.org/jira/browse/BEANUTILS-157>

<sup>11</sup> <https://github.com/apache/commons-beanutils/commit/3a4fa468ab68d9a09e7963c2cd07a8540891cac6>



Commons BeanUtils / **BEANUTILS-157**

## Beanutils's describe() method cannot determine reader methods for anonymous class

**Details**

Type:	Bug	Status:	<b>CLOSED</b>
Priority:	Minor	Resolution:	Fixed
Affects Version/s:	1.6.1	Fix Version/s:	1.8.0
Component/s:	Bean / Property Utils		
Labels:	None		
Environment:	Operating System: Windows XP Platform: PC		
Bugzilla Id:	33461		

**People**

Assignee: Niall Pemberton

Reporter: Thorbjørn Ravn Andersen

Votes: 0 [Vote for this issue](#)

Watchers: 0 [Start watching this issue](#)

**Description**

**Issue Links**

**Activity**

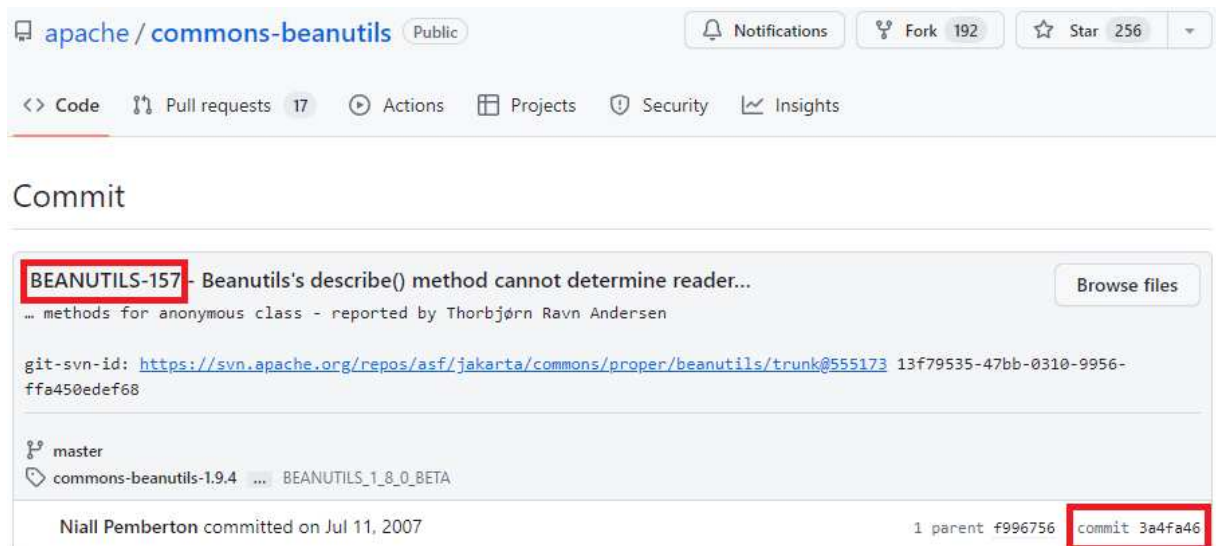
**Dates**

Created: 09/Feb/05 19:38

Updated: 02/May/13 02:29

Resolved: 11/Jul/07 05:45

Figure 4 – Issue BEANUTILS-157, commons-beanutils project, available in Jira.



apache / commons-beanutils Public Notifications Fork 192 Star 256

[Code](#) [Pull requests 17](#) [Actions](#) [Projects](#) [Security](#) [Insights](#)

## Commit

**BEANUTILS-157** - Beanutils's describe() method cannot determine reader... [Browse files](#)

... methods for anonymous class - reported by Thorbjørn Ravn Andersen

git-svn-id: <https://svn.apache.org/repos/asf/jakarta/commons/proper/beanutils/trunk@555173> 13f79535-47bb-0310-9956-ffa450edef68

master

commons-beanutils-1.9.4 ... BEANUTILS\_1\_8\_0\_BETA

Niall Pemberton committed on Jul 11, 2007 1 parent f996756 **commit 3a4fa46**

Figure 5 – Commit 3a4fa46, commons-beanutils project, available in Github.

Additionally, to identify changes at the level of atoms of confusion generated by each commit, BOHR was also run on the immediately preceding system version of each mapped commit. This approach ensures that the dataset contains information about atoms of confusion both before and after each commit mapped in the dataset.

### 3.3 Dataset Description

The dataset comprises information from three perspectives: project management from Jira, code versioning from Git, and atoms of confusion from BOHR. It is organized into Comma Separated Values (CSV) files. Table 5 displays the projects entity of the dataset with general information about the 21 subject projects, like project name, owner, manager, category, Git and Jira repositories, etc.

The entity *Jira Dataset*, identified by issue key, shows the data collected for each project from the issues in the Jira repository, detailing issue description, status, type, identifier, etc. Furthermore, the entity *Issue Developer Comments Dataset* displays the issue developer comments history. Finally, the issue change log has been extracted and is represented by entity *Issue Changelog Dataset*, encompassing every modification made at the issue level by any user.

The issues were linked to the commits that resolve them, as explained in the Subsection 3.2.4. At this point, we mapped the issue key, the commit hash that resolves it, and its immediately preceding commit, as per entity *Issues-Commits Dataset*.

For each commit related to at least one issue, the log information from the version control system was extracted, as specified in entity *Commit Log Dataset*. Then, for each issue key, the data of the related commits was extracted from Git and aggregated, as shown in entity *Git Dataset*.

In the perspective of ACs, as explained in Subsection 3.2.4, we ran the BOHR on the

Table 5 – Projects Dataset

<b>Field</b>	<b>Description</b>
Name	The project's name
Owner	The project's owner
Manager	The committe responsible for the project
Category	The project's domain category
JiraName	The Jira project's identifier
JiraRepository	The Jira repository address
GitRepository	The Git remote repository address
MainBranch	The main Git branch's name for the project
Site	The project's website address
SinceDate	The starting date for mining project information
ToDate	The ending date for mining project information
LastRelease	The last project's release within the SinceDate-ToDate interval
HashLastRelease	The hash of the LastRelease

version of each project corresponding to each mapped commit that resolves at least one issue, as well as on each commit immediately preceding it. The tool generates detailed results, indicating for each occurrence of AC, the class where it occurs, its type, the related source code, and the corresponding line number, as can be observed in entity *BOHR Detailed Dataset*. Additionally, it also provides an aggregate perspective per package of classes, indicating the number of involved classes, the number of analyzed lines of code, the number of occurrences of ACs in these classes, and a breakdown of occurrences by AC types, as displayed in entity *BOHR Aggregated Dataset*.

Still, in the context of ACs, we created two additional datasets to facilitate the analysis of the research questions addressed in this study and provide an aggregated perspective of the changes made to atoms of confusion for each mapped commit in this research. The first provides information about commits that had a direct impact on existing AC, either through a change in the same occurrence line of the atom or through complete removal of the atom, as detailed in entity *Changed/Deleted ACs Dataset*. The second provides a quantitative breakdown before and after each mapped commit, identifying the number of classes, the number of lines of code, and the number of occurrences of atoms by type, among other information before and after the commit, as displayed in entity *Commit ACs Changes Dataset*.

The details for all dataset entities, including their corresponding fields and descriptions, can be found in Appendix A.

### 3.4 Conclusion

The dataset built in this work establishes a solid foundation for this research, providing the necessary infrastructure to address the research questions proposed in the study on the impacts of atoms of confusion in open-source Java projects.

The dataset was created from three perspectives: project management, source code versioning, and atoms of confusion. Data was extracted from 21 open-source Java projects, linking over 8,000 commits, 4,000 reported issues, and 7,000 atoms of confusion from the subject projects.

The entire dataset, along with the methodology and scripts necessary for its creation, has been made available to the community at (PINHEIRO, 2023). In this way, other researchers can validate this research, explore new hypotheses, and expand the dataset with additional data, including new Java projects.

## 4 RESULTS AND DISCUSSION

This chapter presents the results and discussions of this research. Section 4.1 presents the concepts related to the methodology used to obtain the results that are discussed in this chapter. Sections 4.2, 4.3, and 4.4 display the results for each research question proposed in this study. Section 4.5 discusses the results and provides insights into them. Sections 4.6 and 4.7 detail the implications for researchers and practitioners. Ultimately, Section 4.8 presents the concluding remarks for this chapter.

### 4.1 Preliminary Concepts

This section summarizes the methodology used in this research to obtain the results presented in this chapter. Figure 6 depicts the research workflow.

**First**, 21 open-source Java projects were selected as the subject of our empirical investigation, following the criteria outlined in Section 3.2.2.

In the **second step**, our objective was to mine project management information from Jira and code versioning information from Git to form the first two perspectives of our dataset. Then, we aimed to relate which commits resolve which issues for the 21 projects studied in this research. From this association, we can infer the type of a particular commit, as the developers of the studied projects classify each Jira issue in: bug-fix, improvement, new feature, sub-task, task, test, or wish. Table 1 shows their definition. All project data was mined from their inception until May 22, 2023, the date when the dataset was built. To link a commit with a Jira issue, we searched the commit messages to identify the associated issue ID. This strategy, as described by VIEIRA *et al.*, is followed by the ASF projects to maintain traceability between source code modifications and project management changes (VIEIRA *et al.*, 2019).

To complete our dataset with its third perspective (**third step**), we executed the BOHR tool to obtain information about ACs before and after each commit analyzed in this study. During this phase, our primary objective was to address the first research question outlined in Section 1.3. To achieve this, we utilized data about the number of issues reported by type and the number of ACs across the last release of the projects, as we can observe in Table 4.

In the **fourth step**, Python scripts were used to search for the ACs identified in the version immediately preceding each commit. These scripts analyzed the removed and added lines of each commit, thereby identifying the ACs that were removed or changed in each commit,



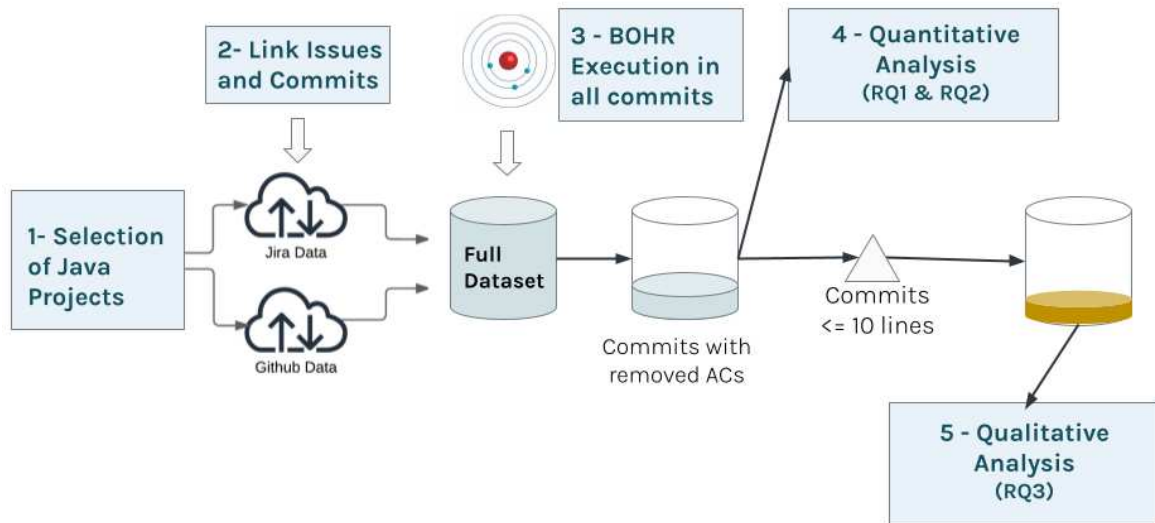


Figure 6 – Research Workflow

thus addressing the second research question outlined in Section 1.3.

Finally, in the **fifth step**, our focus is to select a subset of bug-fix and improvement commits that involve the removal of ACs. This step aimed to deepen our understanding and assess whether the removed ACs can directly contribute to these maintenance tasks. To facilitate manual analysis, we only considered ACs removed from commits with a maximum of ten lines deleted. This decision was made based on the recognition that as the number of lines deleted in a commit increased, determining the specific impact of the AC on the bug or improvement became more challenging, ambiguous, and time-consuming. During manual analysis, we thoroughly examined the commit messages, source code, title, description, and developers' comments related to the corresponding issue. Our objective was to determine if the AC can significantly directly contribute to the bug or improvement. For each AC, we documented whether it likely directly contributed to the occurrence of the maintenance task. Additionally, we specified the evidence that led to our conclusion in the dataset.

#### 4.2 RQ1. To what extent do atoms of confusion relate to the type of maintenance tasks?

**Summary of RQ1:** Maintenance tasks tagged as bug-fix, improvement, and new feature have a positive correlation (Pearson's coefficient  $>0.60$ ) with the number of ACs.

The graph depicted in Fig. 7 illustrates the correlation between the number of issues

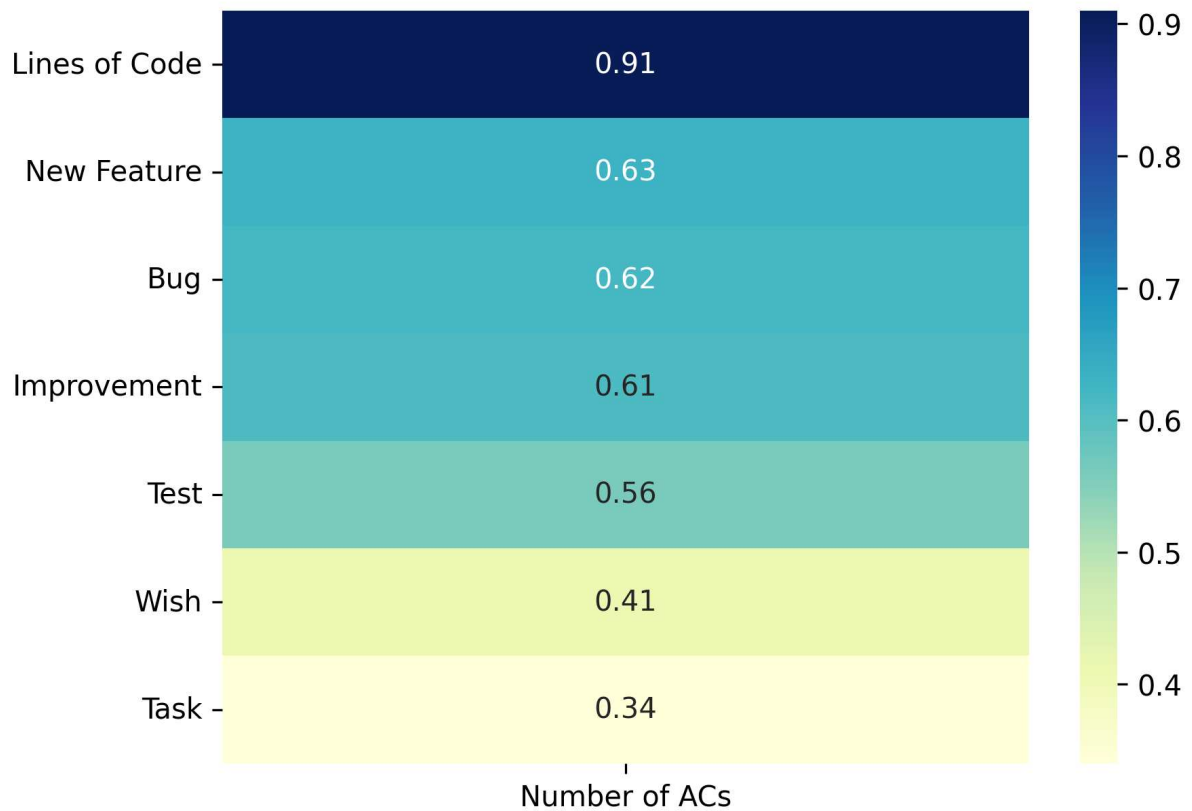


Figure 7 – Number of Atoms of Confusion versus Reported Issues by Type

of a specific type and the number of ACs across the last release at the time the dataset was built of the 21 subject projects, as specified in Table 4.

One can observe that maintenance activities (new feature, bug-fix, and improvement) are the ones that have the highest correlation with the number of ACs, with a Pearson coefficient greater than 0.60. Additionally, we also observe a high correlation between project size, in terms of lines of code, and the number of ACs, as previously shown in other research (MENDES *et al.*, 2022).

Although the number of bugs and improvements increases as the number of ACs grows, we further investigated this phenomenon to assess whether an atom of confusion could likely have a direct impact on the occurrence of a subsequent bug-fix or improvement commit.

#### 4.3 RQ2. In what type of maintenance tasks are atoms of confusion more frequently removed?

**Summary of RQ2:** (i) Commits that remove ACs are rare (4.54%); (ii) The rate of AC removal in bug-fix commits was not higher than the rate of removal in other types of commits in 18 of 21 studied projects.

To answer this question, we analyzed the commits that remove ACs. For that, we counted the ACs removed in all 8,641 commits analyzed. We found **only 391 commits** (4.52%), indicating the rarity of the phenomenon. Bug-fix and improvement commits had higher occurrences with AC removals, 145 and 149 commits, respectively, as shown in Table 6. However, we did not find bug-fix commits removing ACs in eight out of twenty-one projects. In two of them, *exec* and *pro*, neither type of commit removed ACs.

Fig. 8 presents the distribution of commits with at least one removal for each project. It shows how different types of commits contribute to removing ACs across the projects. According to the figure, in a total of 12 projects, improvement commits were the majority in terms of eliminating ACs. However, there were also five projects, such as *io* and *dhcp*, where bug-fix constituted the majority of commits with ACs removed. For example, in the *io* project, out of the 19 commits that removed atoms, 12 of them were categorized as bug-fix.

Fig. 9 displays the number of commits with at least one AC removal. It shows the projects having at least eight commits satisfying this criteria. We observed that the *math*, *lang*, and *compress* projects had the highest number of commits removing ACs. Not by chance, these projects also have the most significant size in lines of code. Our results align with previous research findings that larger projects tend to have more ACs, thus leading to an expectation of more ACs being removed (MENDES *et al.*, 2022).

As aforementioned, bug-fix and improvement commits were the most common types that removed ACs across the projects, accounting for 75.2% of such cases. However, they also represent the most frequent type of commits overall. Therefore, we needed to conduct a more in-depth analysis to determine if bug-fix and improvement commits continue to have the highest proportion of commits involving AC removals.

A shift in the results is observed by normalizing this phenomenon (dividing the number of commits removing ACs by the total number of commits for each type), as illustrated in Fig. 10. Task commits had the highest ratio of commits involving AC removals in eight projects, exactly where task-type commits took place (i.e., we did not find in 13 projects task

Table 6 – Commits With ACs Removals

	<b>Bug-fix</b>	<b>Improv</b>	<b>NewFeat</b>	<b>Task</b>	<b>Others</b>
<b>Total N° of Commits</b>	3,857	2,763	1,161	421	439
<b>Commits with Removals</b>	145	149	40	42	15
<b>Ratio</b>	3.76%	5.39%	3.45%	9.98%	3.42%
<b>Distribution</b>	37.08%	38.11%	10.23%	10.74%	3.84%

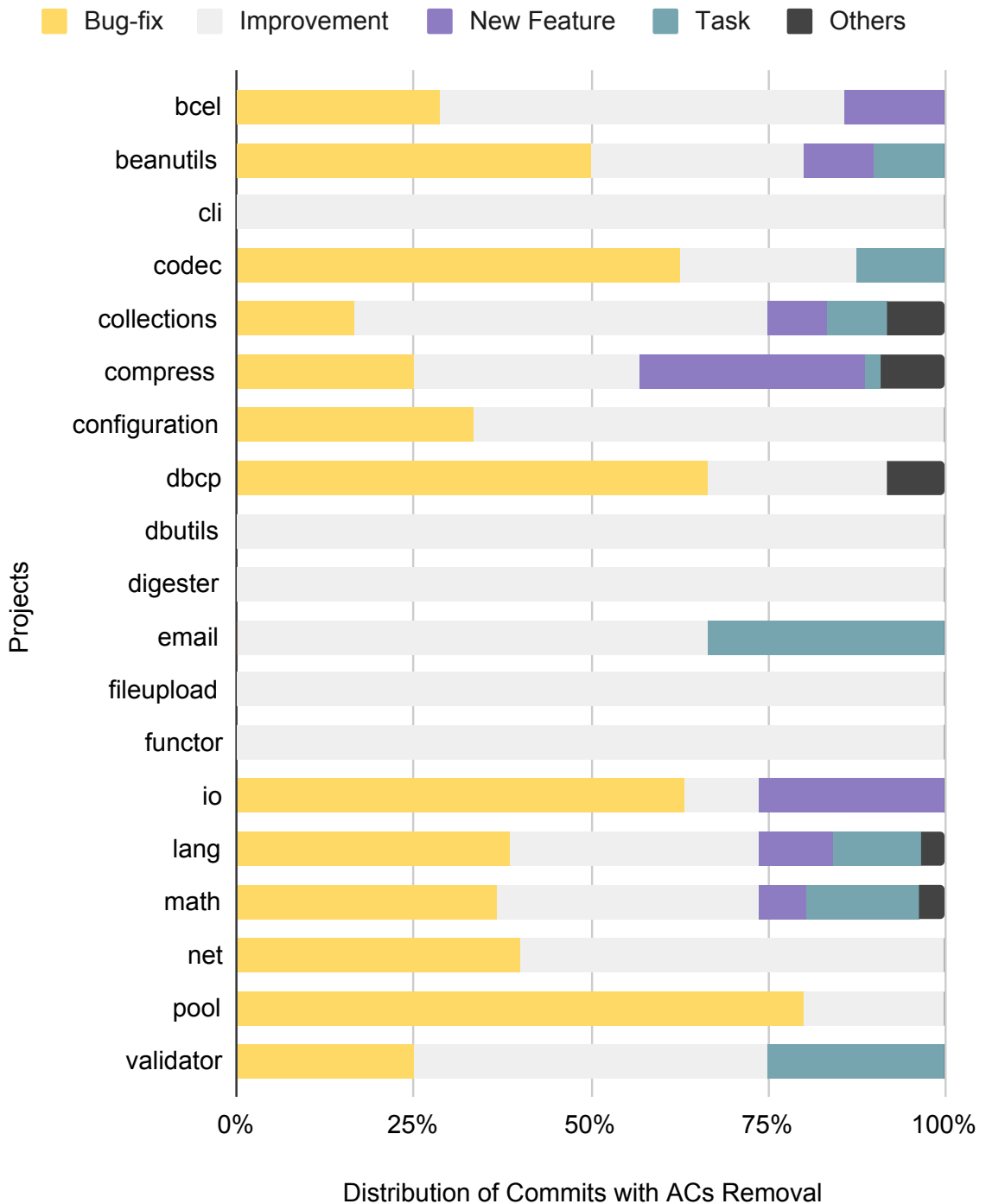


Figure 8 – Distribution of Commits with AC Removals per Project

commits removing ACs). We emphasize that task commits occurred in 20 projects (except for the *digest* project). Regarding improvement and bug-fix commits, the first also had the highest ratio in eight projects. In comparison, bug-fix commits had the highest percentage only in two projects (i.e., *io* and *pool*).

Fig. 10 along with Table 6 reveal that although bug-fix and improvement commits

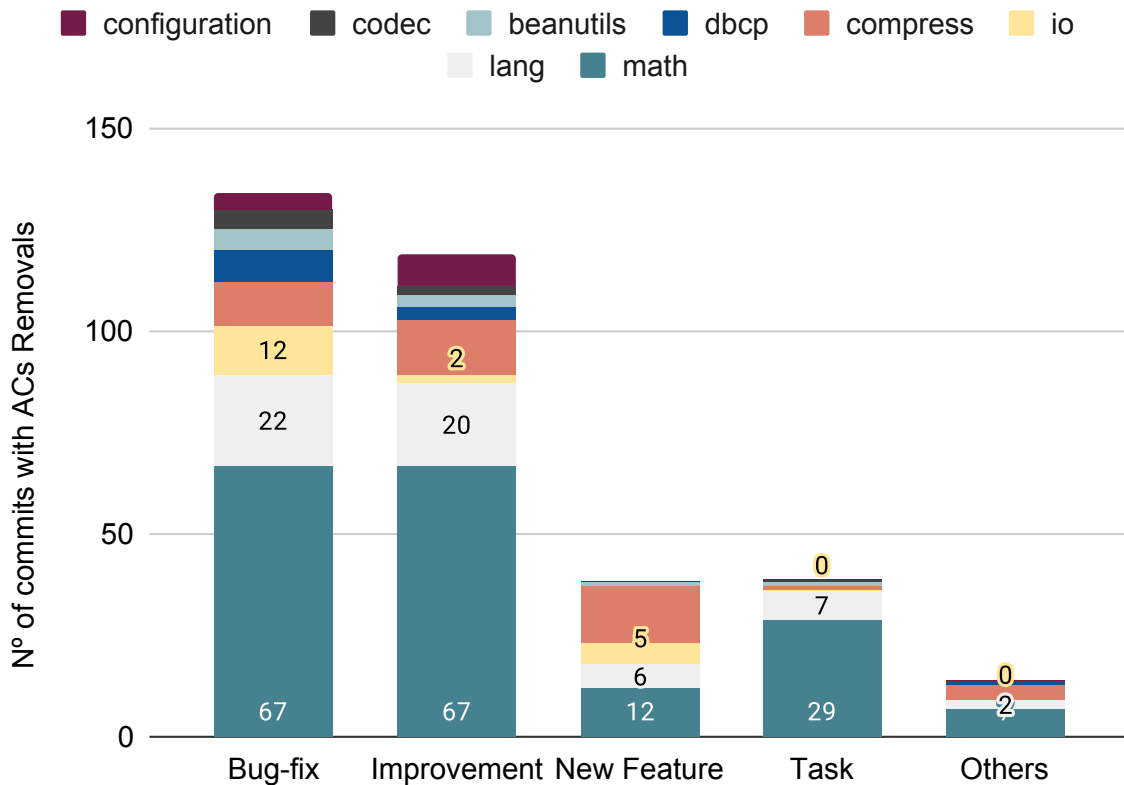


Figure 9 – Projects with more than 8 commits removing ACs

are prominent in absolute terms, their proportional representation in this phenomenon changes when we look at the total number of commits for each type.

In the previous analysis, we focused solely on the number of commits, disregarding that a commit can remove multiple ACs. For this reason, we further pursue our study by checking the number of ACs removed per commit type. Table 7 summarizes the analysis of the number of atoms removed by commit type. We found **2425 atoms removed**. Once again, the improvement (999 out of 2425 ) and bug-fix (756 out of 2425) commits have the highest number of ACs removed. The table also displays the number of ACs removed every 100 commits.

GOPSTEIN *et al.* in (GOPSTEIN *et al.*, 2018) showed that for GCC, bug-fix commits removed atoms at a rate of 1.25x compared to non-bug-fix commits. Fig. 11 shows the results when grouping the commits in the same way for the 21 Java projects.

The mean removal rate per 100 bug-fix commits was 10,59 ( $SD = 2,35$ ), while the mean rate for non-bug-fix commits was 21.4 ( $SD = 2,51$ ). **Only three projects had a higher AC removal rate** in bug-fix commits (i.e., *io*, *codec*, and *beanutils* projects). A Mann-Whitney U test was performed to evaluate whether non-bug-fix commits differed from bug-fix commits regarding the ACs removal rate per 100 commits. The results indicated that

Table 7 – Removed ACs per Commit Type

	<b>Bug-fix</b>	<b>Improv</b>	<b>NewFeat</b>	<b>Task</b>	<b>Others</b>
<b>Total N° Commits</b>	3,857	2,763	1,161	421	439
<b>N° of Removed ACs</b>	756	999	241	408	21
<b>Ratio Removed ACs</b>	31.18%	41.20%	9.94%	16.82%	0.87%
<b>ACs removed every 100 commits</b>	19.60	36.16	20.76	96.91	4.78

non-bug-fix commits had a significantly greater rate of ACs removed than bug-fix commits,  $z = [0.3], p = [0.02713], U = [297]$ .

#### 4.4 RQ3. How often are atoms of confusion likely to directly contribute to a maintenance task?

**Summary of RQ3:** Out of a universe of 8,641 commits from 21 analyzed projects, 391 removed ACs. Among them, 53 met the condition for our qualitative analysis: bug-fix or improvement commits that had up to 10 lines deleted. In 7 of these commits, 9 removed ACs were likely to contribute directly to the occurrence of a bug or improvement.

Up to this point in the research, quantitative analyses had not yielded a strong indication that ACs can directly cause bugs or serve as triggers for improvements. As a result, we decided to further explore cases where atoms of confusion were removed in bug or improvement

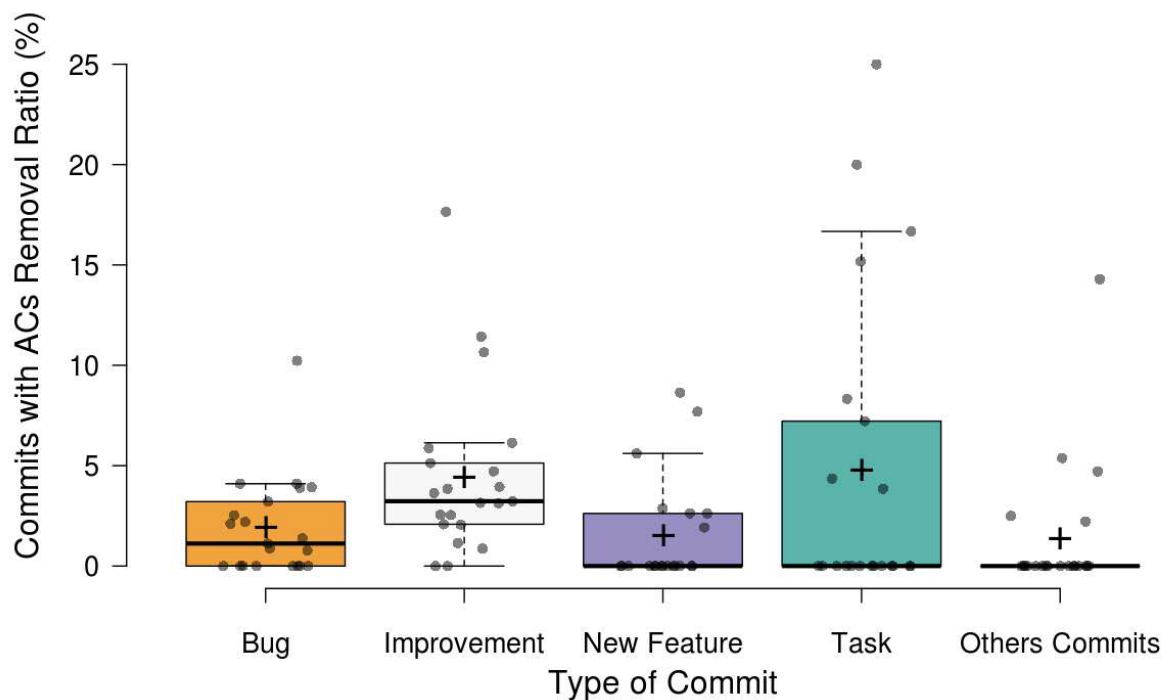


Figure 10 – Ratio of Commits with At Least One AC Removal

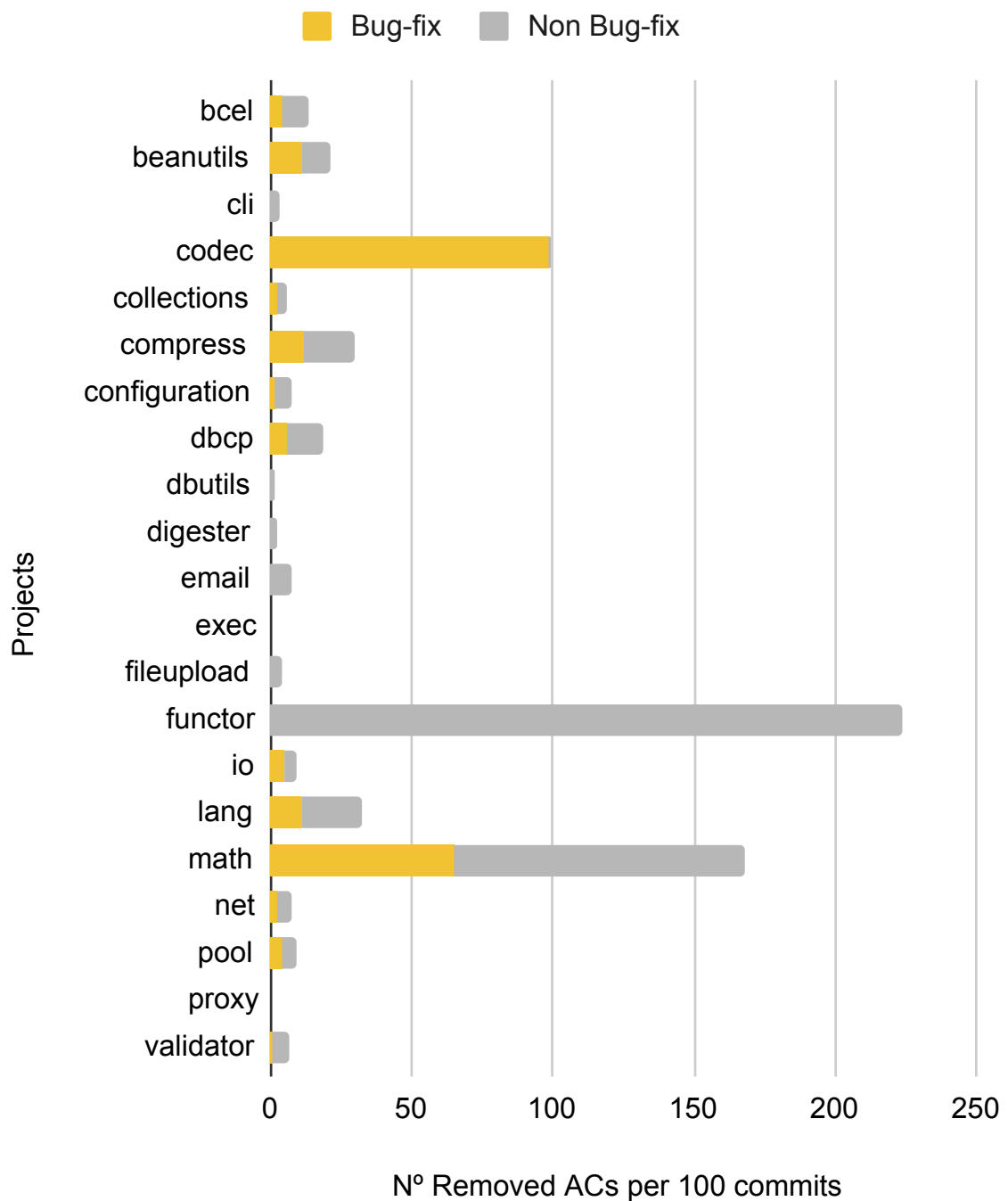


Figure 11 – Bug-fix x Non-Bug-fix Commits

commits, to better understand if such atoms could be directly contributing to the occurrence of such maintenance tasks. According to GOPSTEIN *et al.*, numerous studies on confusion in software systems primarily focus on quantitative definitions and analysis but neglect qualitative research, which can provide deeper insights and enrich the understanding of the topic. The qualitative analysis explores how and why ACs can lead to confusion (GOPSTEIN *et al.*, 2020). With this in mind, during the process of validating and collecting data for our study, we

Table 8 – Atoms of confusion that are likely to be the trigger of a maintenance tasks

<b>Acronym</b>	<b>Jira Issue</b>	<b>Type</b>	<b>Hash</b>	<b>Evidence</b>
IOP	BEANUTILS-351	Bug	c90048ca	Issue Title Issue Description Commit Message
IOP	MATH-318	Bug	83f18d52	Issue Title Issue Description Commit Message
Post-Inc/Dec	COMPRESS-389	Bug	0ee8f1e8	Issue Title Issue Description Commit Message
Pre-Inc/Dec	BCEL-197	Improv.	cf39e4cd	Issue Comments
Pre-Inc/Dec	BCEL-197	Improv.	cf39e4cd	Issue Comments
Pre-Inc/Dec	COMPRESS-453	Bug	e8c44e60	Issue Comments
TC	MATH-153	Bug	409d56d2	Issue Comments Issue Description
TC	MATH-153	Bug	409d56d2	Issue Comments Issue Description
TC	POOL-85	Bug	acb09a57	Issue Title Issue Description Commit Message

encountered instances where ACs were likely to contribute directly to the occurrence of a bug or an improvement. Therefore, we conducted a more in-depth qualitative analysis to investigate this phenomenon further.

As we mentioned in Section 1.3, we only analyzed bug-fix and improvement commits removing ACs with a maximum of ten lines deleted. From the 19 projects with ACs removals, we scrutinized 53 bug-fix and improvement commits. These commits had removed 77 ACs. To achieve this, we investigated the 53 commit messages, over 2394 source code lines, and the 53 descriptions and titles of the linked issues, in addition to 374 developers' comments.

We found 9 ACs that could likely have directly contributed to those maintenance tasks, as displayed in Table 8. Thus, 11.68% of the 77 analyzed ACs could likely have contributed directly to the occurrence of a bug or improvement.

Although some analyzed ACs showed indications of contributing to a maintenance activity, they could not be classified as such because the issue and the commit lacked sufficient information to support such a conclusion. This was the scenario with improvement commit hash 9cc0604<sup>1</sup> described in the Introduction. We did not discover sufficient information on the commit message and the issue's title, description, or comment to substantiate our suspicion.

<sup>1</sup> <https://github.com/apache/commons-compress/commit/9cc0604>



Hence, we excluded it from the computation of the 9 ACs listed in Table 8.

We only counted cases that the contribution of the AC was clear based on the information reported by the developers in the change control and project management systems. We wrote one example of finding in the Introduction (i.e., the bug-fix commit hash c90048ca<sup>2</sup>). Below, we detail other examples.

The commit hash acb09a57<sup>3</sup> in the commons-pool can illustrate how we classify whether an AC could likely have contributed to a maintenance task. This commit deleted a single line of code that contained the Type Conversion Atom (see Fig. 12). This AC occurs when there is a conversion from a larger data type to a smaller one, which can result in a loss of precision and unexpected results for the programmer. From the commit message, highlighted in Fig. 13, we observed that the cause of the reported bug in Jira issue POOL-85<sup>4</sup> was directly related to the information loss caused by the atom that was removed.

```

@@ -1451,7 +1451,15 @@ public int compareTo(Object obj) {
1451 1451     }
1452 1452
1453 1453     public int compareTo(ObjectTimestampPair other) {
1454 -         return (int) (this.timestamp - other.timestamp);
1454 +         final long timestampdiff = this.timestamp - other.timestamp;
1455 +         if (timestampdiff == 0) {
1456 +             // make sure the natural ordering is consistent with equals
1457 +             // see java.lang.Comparable Javadocs
1458 +             return System.identityHashCode(this) - System.identityHashCode(other);
1459 +         } else {
1460 +             // handle int overflow
1461 +             return (int) Math.min(Math.max(timestampdiff, Integer.MIN_VALUE), Integer.MAX_VALUE);
1462 +         }
1455 1463     }
1456 1464 }
1457 1465

```

Figure 12 – Commit acb09a57, commons-pool project, available in Github

Another example is the improvement commit hash cf39e4cd<sup>5</sup> in the commons-bcel, which deleted only two lines of code, and both of them contained the Pre Increment/Decrement Atom, as can be seen in the Fig. 14. This AC involves the use of pre-increment/decrement unary operators. The pre-increment/decrement unary operator both increments/decrements the associated variable and returns the result of the expression. Due to unfamiliarity with this

<sup>2</sup> <https://github.com/apache/commons-beanutils/commit/c90048ca>

<sup>3</sup> <https://github.com/apache/commons-pool/commit/acb09a57>

<sup>4</sup> <https://issues.apache.org/jira/browse/POOL-85>

<sup>5</sup> <https://github.com/apache/commons-bcel/commit/cf39e4cd>

**Make sure the natural ordering is consistent with equals,** [Browse files](#)  
 see `java.lang.Comparable` Javadocs,  
 fixes P00L-85 as reported by Mike Martin.  
 Added checks in case the long to int cast overflows.

git-svn-id:  
<https://svn.apache.org/repos/asf/jakarta/commons/proper/pool/trunk@468834>  
 13f79535-47bb-0310-9956-ffa450edef68

---

 master  
 rel/commons-pool-2.11.1 ... POOL\_1\_4\_RC1

---

**Sandy McArthur Jr** committed on Oct 29, 2006

1 parent [43f6578](#) commit [acb09a5](#)

Figure 13 – Commit Message `acb09a57` , commons-pool project, available in Github

operator, doubts about its functionality can arise, leading to confusion. Furthermore, another potential source of confusion is the similarity between the pre-increment/decrement operator and the post-increment/decrement operator, which only returns the variable's value without modifying it. In this case, the confusion caused by the AC becomes clear in the developers' comments, as we can observe in Fig. 15.

```

src/main/java/org/apache/commons/bcel6/classfile/Utility.java
@@ -861,10 +861,10 @@ public static String signatureToString( String signature, boolean chopit ) {
861 861 // check for wildcards
862 862 if (signature.charAt(consumed_chars) == '+') {
863 863     type = type + "? extends ";
864 864     consumed_chars = ++consumed_chars;
864 864     consumed_chars++;
865 865 } else if (signature.charAt(consumed_chars) == '-') {
866 866     type = type + "? super ";
867 867     consumed_chars = ++consumed_chars;
867 867     consumed_chars++;
868 868 } else if (signature.charAt(consumed_chars) == '*') {
869 869     // must be at end of signature
870 870     if (signature.charAt(consumed_chars + 1) != '>') {

```

Figure 14 – Commit `cf39e4cd`, commons-bcel project, available in Github

---

▼ Ⓞ Sebb added a comment - 13/Aug/15 19:34

There is some very odd code in the patch that was applied.

There are two instances of

```
consumed_chars = ++consumed_chars;
```

What is this supposed to be doing?

---

▼ Ⓞ Mark Roberts added a comment - 13/Aug/15 20:07

I can't be sure without you providing a bit more context - I do see two instances, but they are in different if blocks. One is part of the "+" case, the other is for the "-" case.

(Light bulb) - I bet you are asking about the "+foo" construct itself? I guess it's a result of learning C/C+ before Java. My fingers wanted to type

```
consumed_chars++;
```

but, of course, that's no good in Java. I agree "foo=++foo;" looks a little strange and I have no problem if you wish to change it to the equivalent, but more traditional, "foo=foo+1;"

Thanks, Mark

---

▼ Ⓞ Sebb added a comment - 13/Aug/15 21:53

What is wrong with

```
consumed_chars++;
```

That has been in Java since it was an acorn, surely?

Certainly Eclipse is happy with it...

---

Figure 15 – Issue BCEL-197 , commons-bcel project, available in Jira

## 4.5 Results Discussion

The primary objective of this research was to investigate the effects of ACs on real-world Java systems. We observed a positive correlation between the number of maintenance tasks in a project and the presence of atoms of confusion (**RQ1**). This correlation is more robust with bug-fix and improvement commits. We also identified a similar pattern with lines of code. These correlations, however, seem to be caused by the project's scale, wherein larger projects

tend to have a greater quantity of ACs as well as bug-fix and improvement tasks.

Our subsequent study (**RQ2**) aimed to determine, through quantitative analysis, whether ACs are more commonly removed in bug-fix commits compared to other types of commits. However, our findings diverged significantly from those reported in the study conducted by GOPSTEIN *et al.* for the GCC project. Initially, bug-fix commits did not exhibit a prominent status regarding the absolute number of AC removals or occurrences compared to other commit types. When comparing the removal rate of ACs in bug-fix commits versus non-bug-fix commits, we observed that ACs were more frequently removed in non-bug-fix commits in most analyzed projects (18 out of 21). Surprisingly, improvement commits emerged as the category with the highest number of AC removals (999) and occurrences (149 compared to 145 for bug-fix commits). Even in terms of percentage, 5.49% of improvement commits removed at least one AC, while the corresponding rate for bug-fix commits was 3.76%.

A bug-fix commit is intended to address and resolve a bug. In contrast, an improvement commit aims to enhance the system with a potential consequence of bug prevention. For this reason, we decided to redo our RQ2 analysis by comparing the removal of ACs between bug-fix and improvement commits against other commit types.

As depicted in Figure 16, the removal rate of ACs exhibited a proportionally higher trend in bug-fix and improvement commits across 14 of the 19 projects where AC removals occurred. A Mann-Whitney U test was performed to evaluate whether bug-fix and improvement commits differed from other commits regarding the ACs removal rate per 100 commits. The results indicated that bug-fix and improvement commits had a significantly greater rate of ACs removed than other commits,  $z = [0.4]$ ,  $p = [0.002936]$ ,  $U = [328]$ .

This result implies that attention should also be given to improvement-type commits. In response to RQ3, our qualitative analysis corroborates this insight, as two improvement commits had the removal of ACs as their likely trigger cause.

Commenting further on the qualitative analysis, it is worth noting that the 9 ACs found were of four types, namely: 3 Infix Operator Precedence, 1 Post Increment Decrement, 3 Pre Increment Decrement, and 3 Type Conversion.

Overall, Infix Operator Precedence was the most removed atom (1,410), followed by Conditional Operator (361). Type Conversion (343) and Post Increment Decrement (230). The others had less than 41 removals. For instance, Pre Increment Decrement occurred in 13 removals. Accordingly, despite being one of the least removed ACs, Pre Increment Decrement

was likely the cause of one bug-fix and two improvement commits.

According to MENDES *et al.* such types of ACs are precisely the most prevalent, second only to the Conditional Operator and Logic as Control Flow (MENDES *et al.*, 2022). These findings indicate that some ACs may deserve more attention during development as they are more likely to impact future maintenance tasks.

#### 4.6 Implications for Researchers

As previously mentioned, Gopstein *et al.* (2017) introduced the concept of *Atom of Confusion (AC)*. Previous research has shown that ACs have been identified as significant and prevalent in C and C++ projects and can cause issues in code comprehension, hinder software maintenance, and introduce challenges in software evolution in these projects (GOPSTEIN *et al.*, 2017; GOPSTEIN *et al.*, 2018).

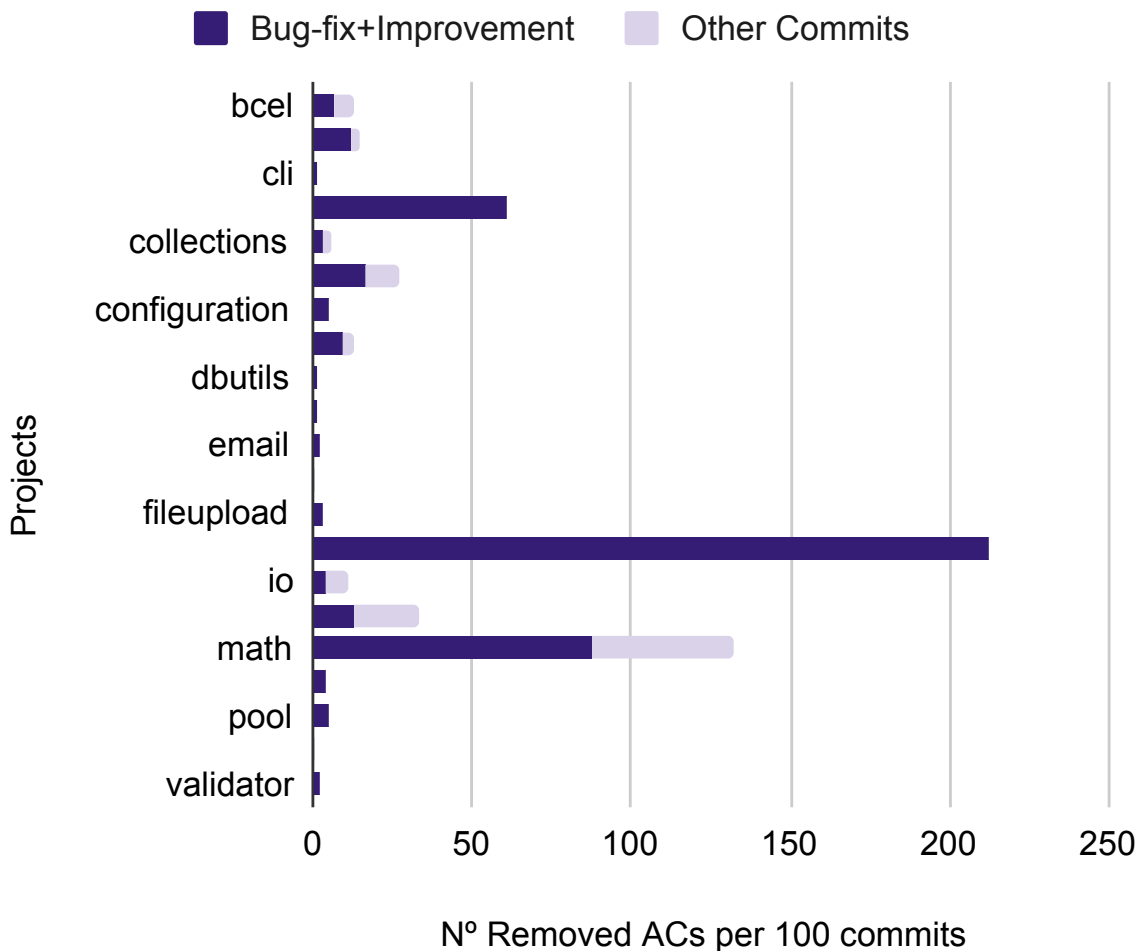


Figure 16 – Bug-fix and Improvement x All Other Commits

In this regard, MENDES *et al.* showed that ACs in Java projects were also significantly prevalent, and the number of their occurrences increased over time. However, to the best of our knowledge, there is still no study that assesses the real impacts of ACs in open-source Java projects.

Thus, aiming to measure the potential impacts of ACs in open-source Java projects, we studied the relationship between the prevalence of ACs and subsequent occurrences of bugs and triggers for improvements. As seen in the previous section, we obtained quantitative results showing correlations between ACs and maintenance tasks, along with qualitative results listing practical examples from real-world Java projects. In these examples, we mapped developer comments, commit messages, issue titles, and descriptions suggesting that a particular atom of confusion may have directly contributed to a bug or improvement. We believe our findings motivate researchers to further investigate the presence and impacts of ACs in Java projects.

Beyond the experimental results of this work, we also provide essential infrastructure for future research. We built a dataset linking over 8,000 commits, 4,000 reported issues, and 7,000 ACs from 21 Java projects, aggregating information from three distinct perspectives: project management, code versioning, and atoms of confusion. This dataset enables future studies to continue exploring the impacts of ACs in Java projects. Additionally, we have made available all the necessary scripts for creating the dataset, which can be used to aggregate new information and expand it with additional Java projects. Links to all our materials are available at (PINHEIRO, 2023).

#### **4.7 Implications for Practitioners**

Previous studies have shown that atoms of confusion can cause serious difficulties during code comprehension: a task performed at every stage of the software lifecycle. This can negatively impact the productivity of a software development team, potentially leading to delays in the schedule and an increase in the budget of a software project (MAALEJ *et al.*, 2014; GOPSTEIN *et al.*, 2017).

Although there are already studies showing the significant prevalence of ACs in real-world systems and their actual impact on confusing developers, they still do not give due importance to the issue, as observed by (MEDEIROS *et al.*, 2019). In this research, pull requests to popular and relevant open-source projects were submitted, simply replacing a confusing code pattern with an equivalent, non-confusing code. Only 22% were accepted, while the others were

rejected with the justification that the code was functioning without errors, or they were not even responded to (MEDEIROS *et al.*, 2019).

This phenomenon may be occurring because we still have few results showing the real impacts of using such code patterns in real-world systems. Therefore, addressing this issue and being able to demonstrate how the use of ACs can negatively impact a software project becomes important to motivate developers to avoid and refactor such code patterns to improve code readability and prevent future issues in code comprehension, and consequently, in the schedule and budget.

## 4.8 Conclusion

In conclusion, this chapter presented a comprehensive exploration of the results and discussions arising from the investigation into the effects of Atoms of Confusion (ACs) on 21 open-source Java projects. The research aimed to address three key research questions, each shedding light on distinct aspects of ACs' impact on maintenance tasks and code comprehension.

In **RQ1**, the findings revealed a positive correlation (Pearson's coefficient  $> 0.60$ ) between the number of ACs and the occurrence of maintenance tasks (bug-fix, improvement, and new feature).

To better understand this correlation and deepen our analysis of the contributions of ACs to the occurrence of maintenance tasks, in **RQ2**, we analyzed the context in which ACs are removed in different types of commits throughout a Java project. To achieve this, we compared the rate of removed ACs in each type of commit across 21 open-source Java projects. By examining the version control histories for each reported bug, we could identify the modified code aimed at resolving the issue. After all, code removed in a bug-fix commit is more likely to have directly contributed to the bug than code removed in other types of commits (GOPSTEIN *et al.*, 2018). Our study revealed that commits removing ACs are rare, accounting for only 4.54%, and the rate of AC removal in bug-fix commits was not higher than in other types for 18 out of the 21 projects studied.

However, surprisingly, improvement commits emerged as the category with the highest number of AC removals. While a bug-fix commit is intended to address and resolve a bug, an improvement commit aims to enhance the system with the potential consequence of preventing bugs. For this reason, we decided to revisit our **RQ2** analysis by comparing the removal of ACs between bug-fix and improvement commits against other commit types. The

removal rate of ACs exhibited a proportionally higher trend in bug-fix and improvement commits across 14 of the 19 projects where AC removals occurred. This result implies that attention should also be given to improvement-type commits.

To support the quantitative analysis conducted in this study, in **RQ3**, we conducted a qualitative analysis. We chose to investigate 53 bug-fix and improvement commits that had a maximum of 10 lines removed. We scrutinized 53 commit messages, over 2394 source code lines, and the 53 descriptions and titles of the linked issues, in addition to 374 developers' comments. Following this, out of the 77 analyzed atoms of confusion, we identified 9 (11.68%) that have a high probability of having directly contributed to the occurrence of a bug or improvement.

In conclusion, our study contributes to the evolving understanding of ACs by bridging the gap in research on their impacts on open-source Java projects. Moreover, beyond the immediate experimental outcomes, we offer a valuable foundation for future research endeavors. The comprehensive dataset we compiled serves as a robust resource for researchers exploring the impacts of ACs in Java development. By providing essential infrastructure and scripts, we facilitate the replication of our study and encourage the aggregation of additional data from diverse Java projects. Despite existing research that demonstrates the substantial prevalence of ACs in real-world systems and their tangible impact on developer confusion, the development team still does not give due importance to the issue (MEDEIROS *et al.*, 2019). This occurrence might be taking place due to the limited number of results demonstrating the actual impacts of employing such code patterns in real-world systems. Addressing this gap becomes imperative to motivate developers to avoid and refactor ACs, enhancing code readability and preventing future comprehension issues. By demonstrating the negative impact of ACs on software projects, our research advocates for a heightened awareness of these issues among practitioners, promoting proactive measures for improved code quality, comprehension, and overall project success.



## 5 CONCLUSION

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

### 5.1 Final Considerations

This study investigated the possible impacts of atoms of confusion on the software development lifecycle of 21 open-source long-lived Java projects. In our analysis, our results showed that there is a positive correlation between the number of ACs and the number of reported bugs and improvements in a project (answering RQ1).

Additionally, we have better understood the context in which ACs were removed in distinct commit types. First, we observed that commits removing at least one AC are rare (4.54%). Also, we observed that in 14 out of 19 projects having commits removing ACs, there was a higher rate of atom removal in bug-fix and improvement commits compared to the other types of commits (answering RQ2). Such a phenomenon may indicate that the more ACs there are, the greater the number of bugs and improvements in a project contribute to augmenting the number of maintenance tasks.

To support these conclusions, we conducted a qualitative analysis to assess, based on information reported by developers, whether ACs were likely to be the direct cause of a bug or improvement. Of the analyzed cases, 11.68% were found to likely have directly contributed to the maintenance task (answering RQ3). This study can assist developers in avoiding the inclusion of atoms of confusion in their source code, as it can potentially lead to difficulties in code comprehension during software maintenance and evolution.

### 5.2 Main Contributions

The main contributions of this work are summarized below:

- **Dataset:** we created a repository containing information on project management, code versioning, and atoms of confusion from 21 open-source Java projects, encompassing over 4,000 issues, 8,000 commits, and 7,000 ACs. Available at (PINHEIRO, 2023).
- **Methodology and Source Code:** all the processes and source code required

to create the dataset have been detailed so that researchers can expand it. The scripts and complementary material can be found on the dataset website.

- **Impacts of ACs in Java Projects:** a study of the relationship between ACs and maintenance tasks in 21 open-source Java projects.

In addition, a paper was published along the development of this work:

- O. Pinheiro, L. Rocha, and W. Viana, "How They Relate and Leave: Understanding Atoms of Confusion in Open-Source Java Projects" 2023 **International Working Conference on Source Code Analysis and Manipulation (SCAM)**, 2023.

Furthermore, I co-authored two papers on the topic of this dissertation:

- D. Tabosa, O. Pinheiro, W. Viana, and L. Rocha, "A Dataset of Atoms of Confusion in the Android Open Source Project" 2024 **Mining Software Repositories (MSR)**, 2024.
- 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.

### 5.3 Threats to Validity

We investigated the threats to the validity of our research using the threats classification presented by WOHLIN *et al.*: conclusion, internal, construct, and external validity (WOHLIN *et al.*, 2012).

#### 5.3.1 Conclusion Validity

Threats to the conclusion validity are concerned with factors that impact the capacity to make accurate inferences about the relationship between the treatment and the outcome of an experiment. To mitigate this threat, we investigated known research questions for open-source Java projects that have already been used in previous studies on the impacts of ACs in C and C++ real-world systems (GOPSTEIN *et al.*, 2018). Furthermore, we carefully chose proper statistical tests and correlation measures (Mann-Whitney and Pearson Correlation Coefficient), that have been investigated and validated in previous studies (KUMAR *et al.*, 2011; JI *et al.*, 2009;

STANTON, 2001). Additionally, we have observed the assumptions (e.g., sample distribution, dependence, and size) of our statistical tests, trying to avoid wrong conclusions.

### **5.3.2 Internal Validity**

Internal validity threats can potentially influence the independent variable's causality without the researcher's knowledge. As a result, they threaten the inference of a potential causal link between treatment and outcome. We conducted a qualitative analysis to mitigate this threat to enhance the quantitative findings. In our qualitative analysis, in addition to examining the commit message and source code, we also inspected the title, description, and developers' comments on the related issue. Therefore, to determine if an atom of confusion was likely to be the cause of a bug or improvement, we needed a wealth of information and discussions in each issue/commit as evidence. During this task, we encountered removals of ACs that appeared to have influenced the bug-fix/improvement when analyzing the involved source code. However, in our results, they were not considered as a probable cause of the maintenance task due to a lack of higher-quality information to support this conclusion. Furthermore, although this analysis was primarily conducted by a single researcher, the more complex cases were reviewed by two additional researchers for increased confidence in determining whether the atom was likely to be the cause of a bug or improvement.

### **5.3.3 Construct Validity**

Construct validity concerns generalizing the result of the research to the concept or theory behind the study. We employed a peer debriefing strategy to validate the research design and document review. The aim was to prevent discrepancies in result interpretation. Furthermore, we used an automated detection tool to address our research questions, which has been used in previous studies and had its precision and recall tested with excellent results (MENDES *et al.*, 2021; MENDES *et al.*, 2022). Finally, to perform data mining of the Git and Jira repositories, we used tools that have already been validated and used in other research: PyDriller and Jira Python API (SPADINI *et al.*, 2018; JIRA-PYTHON-LIBRARY, 2012).

### 5.3.4 External Validity

Threats to external validity are conditions that limit the capability to generalize the results of our research to industrial practice. One of the phases of our study involved linking project commits with their respective Jira issues. To avoid external validity on this association, we employed a strategy widely used in previous empirical studies on Apache projects: the ASF project developers specify the issue ID in the messages of the commits that were necessary to resolve the issue (VIEIRA *et al.*, 2019).

The project selection that was the subject of this study was based on existing work on the prevalence of ACs in Java systems, using the BOHR tool (MENDES *et al.*, 2022). As this previous study demonstrated the significant occurrence of ACs in real-world Java projects, we decided to start from this point to show the possible impacts that such prevalence can cause. For this purpose, 21 of the 27 projects addressed in the previous work were chosen. This is because these 21 projects used the same project management and version control tools, which facilitated the construction of our solution to conduct our analysis. The selected projects represent a single domain of real-world Java systems, which can be a threat to external validity. Therefore, as an evolution of this study, we propose to expand the selected projects as the research object, similar to what GOPSTEIN *et al.* did in their research, in which 14 projects were selected, two from each of the following seven domains: operating system, browser, compiler, database, version control, text editor, and web server (GOPSTEIN *et al.*, 2018).

## 5.4 Future Work

Based on this research, some questions arose that we believe are essential and can be the subject of future work. One possible approach is the evolution of the research questions proposed, regarding a more detailed study based on the individual analysis by type of AC. Another aspect to be explored is expanding this study to open-source Java projects from other domains (e.g., mobile, e-commerce), increasing the number of subject projects.

Another aspect that can be studied is the creation of new metrics to evaluate the relationships between atoms of confusion and maintenance activities, such as:

- Are classes with more reported bugs and improvements the ones that have more ACs?
- What is the distance, in lines of code, between ACs and the changes made by bug-fix and improvement commits?

- Are releases with a higher number of ACs also the ones with the highest number of reported bugs and code refactoring?
  - Are the commits with a higher risk of changes the ones with the highest incidence of ACs?
- The delta-maintainability metric is the proportion of low-risk change in a commit and can be measured using the Open Source Delta Maintainability Model (OS-DMM). (BIASE *et al.*, 2019).

## BIBLIOGRAPHY

- AL-ZUBAIDI, W. H. A.; DAM, H. K.; GHOSE, A.; LI, X. Multi-objective search-based approach to estimate issue resolution time. In: **Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering**. [S. l.: s. n.], 2017. p. 53–62.
- ALWIS, B. D.; SILLITO, J. Why are software projects moving from centralized to decentralized version control systems? In: IEEE. **2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering**. [S. l.], 2009. p. 36–39.
- BANKER, R. D.; DAVIS, G. B.; SLAUGHTER, S. A. Software development practices, software complexity, and software maintenance performance: A field study. **Management science, INFORMS**, v. 44, n. 4, p. 433–450, 1998.
- BENNETT, K. H.; RAJLICH, V. T.; WILDE, N. Software evolution and the staged model of the software lifecycle. In: **Advances in Computers**. [S. l.]: Elsevier, 2002. v. 56, p. 1–54.
- BIASE, M. di; RASTOGI, A.; BRUNTINK, M.; DEURSEN, A. van. The delta maintainability model: Measuring maintainability of fine-grained code changes. In: IEEE. **2019 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [S. l.], 2019. p. 113–122.
- BOGACHENKOVA, V.; NGUYEN, L.; EBERT, F.; SEREBRENIK, A.; CASTOR, F. Evaluating atoms of confusion in the context of code reviews. In: IEEE. **2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S. l.], 2022. p. 404–408.
- 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.
- CROWSTON, K.; WEI, K.; HOWISON, J.; WIGGINS, A. Free/libre open-source software development: What we know and what we do not know. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 44, n. 2, p. 1–35, 2008.
- EBERT, F.; CASTOR, F.; NOVIELLI, N.; SEREBRENIK, A. Confusion in code reviews: Reasons, impacts, and coping strategies. In: IEEE. **2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)**. [S. l.], 2019. p. 49–60.
- FJELDSTAD, R. K. Application program maintenance study. **Report to Our Respondents, Proceedings GUIDE**, v. 48, 1983.
- FREY, T.; GELHAUSEN, M.; SAAKE, G. Categorization of concerns: a categorical program comprehension model. In: **Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools**. [S. l.: s. n.], 2011. p. 73–82.
- GERMAN, D. M. The gnome project: a case study of open source, global software development. **Software Process: Improvement and Practice**, Wiley Online Library, v. 8, n. 4, p. 201–215, 2003.
- 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: **Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S. l.: s. n.], 2020. p. 605–616.

- 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**. [S. l.: s. n.], 2017. p. 129–139.
- GOPSTEIN, D.; ZHOU, H. H.; FRANKL, P.; CAPPOS, J. Prevalence of confusing code in software projects: Atoms of confusion in the wild. In: **Proceedings of the 15th International Conference on Mining Software Repositories**. [S. l.: s. n.], 2018. p. 281–291.
- JERMAKOVICS, A.; SILLITTI, A.; SUCCI, G. Mining and visualizing developer networks from version control systems. In: **Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering**. [S. l.: s. n.], 2011. p. 24–31.
- JI, C.; CHEN, Z.; XU, B.; WANG, Z. A new mutation analysis method for testing java exception handling. In: IEEE. **2009 33rd Annual IEEE International Computer Software and Applications Conference**. [S. l.], 2009. v. 2, p. 556–561.
- JIRA-PYTHON-LIBRARY. 2012. <https://jira.readthedocs.io/> [Accessed: (nov. 2022)].
- KO, A. J.; DELINE, R.; VENOLIA, G. Information needs in collocated software development teams. In: IEEE. **29th International Conference on Software Engineering (ICSE'07)**. [S. l.], 2007. p. 344–353.
- KUMAR, K.; GUPTA, P.; PARJAPAT, R. New mutants generation for testing java programs. In: SPRINGER. **Computer Networks and Information Technologies: Second International Conference on Advances in Communication, Network, and Computing, CNC 2011, Bangalore, India, March 10-11, 2011. Proceedings 2**. [S. l.], 2011. p. 290–294.
- LANGHOUT, C.; ANICHE, M. Atoms of confusion in java. In: IEEE. **2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)**. [S. l.], 2021. p. 25–35.
- LEWIS, R.; MELLO, C. A.; ZHUANG, Y.; YEH, M. K.-C.; YAN, Y.; GOPSTEIN, D. Rough sets: Visually discerning neurological functionality during thought processes. In: SPRINGER. **International Symposium on Methodologies for Intelligent Systems**. [S. l.], 2018. p. 32–41.
- MAALEJ, W.; TIARKS, R.; ROEHM, T.; KOSCHKE, R. On the comprehension of program comprehension. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 23, n. 4, p. 1–37, 2014.
- 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 Software Engineering**, Springer, v. 24, n. 4, p. 1693–1726, 2019.
- MENDES, W.; PINHEIRO, O.; SANTOS, E.; ROCHA, L.; VIANA, W. Dazed and confused: Studying the prevalence of atoms of confusion in long-lived java libraries. In: IEEE. **2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S. l.], 2022. p. 106–116.
- MENDES, W.; VIANA, W.; ROCHA, L. Bohr-uma ferramenta para a identificação de átomos de confusão em códigos java. In: SBC. **Anais do IX Workshop de Visualização, Evolução e Manutenção de Software**. [S. l.], 2021. p. 41–45.
- MINELLI, R.; MOCCI, A.; LANZA, M. I know what you did last summer-an investigation of how developers spend their time. In: IEEE. **2015 IEEE 23rd International Conference on Program Comprehension**. [S. l.], 2015. p. 25–35.

- MOCKUS, A. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In: IEEE. **2009 6th IEEE International Working Conference on Mining Software Repositories**. [S. l.], 2009. p. 11–20.
- MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are java software developers using the elipse ide? **IEEE software**, IEEE, v. 23, n. 4, p. 76–83, 2006.
- 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**. [S. l.: s. n.], 2020. p. 243–252.
- 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. Available in: <https://hal.archives-ouvertes.fr/hal-01078532/document>.
- PINHEIRO, O. **Replication Package - How They Relate and Leave: Understanding Atoms of Confusion in Open-Source Java Projects**. Zenodo, 2023. Available in: <https://doi.org/10.5281/zenodo.8105813>.
- RAHMAN, A. Comprehension effort and programming activities: related? or not related? In: **Proceedings of the 15th International Conference on Mining Software Repositories**. [S. l.: s. n.], 2018. p. 66–69.
- RAJLICH, V.; WILDE, N. The role of concepts in program comprehension. In: IEEE. **Proceedings 10th International Workshop on Program Comprehension**. [S. l.], 2002. p. 271–278.
- RATH, M.; MÄDER, P. The seoss 33 dataset—requirements, bug reports, code history, and trace links for entire projects. **Data in brief**, Elsevier, v. 25, p. 104005, 2019.
- ROBILLARD, M. P.; COELHO, W.; MURPHY, G. C. How effective developers investigate source code: An exploratory study. **IEEE Transactions on software engineering**, IEEE, v. 30, n. 12, p. 889–903, 2004.
- SCHRÖTER, I.; KRÜGER, J.; SIEGMUND, J.; LEICH, T. Comprehending studies on program comprehension. In: IEEE. **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. [S. l.], 2017. p. 308–311.
- SINGER, J.; LETHBRIDGE, T.; VINSON, N.; ANQUETIL, N. An examination of software engineering work practices. In: **CASCON First Decade High Impact Papers**. [S. l.: s. n.], 2010. p. 174–188.
- SPADINI, D.; ANICHE, M.; BACCHELLI, A. Pydriller: Python framework for mining software repositories. In: \_\_\_\_\_. **The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)**. [S. l.: s. n.], 2018.
- SPINELLIS, D. Version control systems. **IEEE software**, IEEE, v. 22, n. 5, p. 108–109, 2005.
- STANTON, J. M. Galton, pearson, and the peas: A brief history of linear regression for statistics instructors. **Journal of Statistics Education**, Taylor & Francis, v. 9, n. 3, 2001.



TORRES, A.; OLIVEIRA, C.; OKIMOTO, M.; MARCÍLIO, D.; QUEIROGA, P.; CASTOR, F.; BONIFÁCIO, R.; CANEDO, E. D.; RIBEIRO, M.; MONTEIRO, E. An investigation of confusing code patterns in javascript. **Journal of Systems and Software**, Elsevier, v. 203, p. 111731, 2023.

VIEIRA, R.; SILVA, A. da; ROCHA, L.; GOMES, J. P. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects. In: **Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering**. [S. l.: s. n.], 2019. p. 80–89.

WEINBERG, G. M. **The psychology of computer programming**. [S. l.]: Van Nostrand Reinhold New York, 1971. v. 29.

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**, IEEE, v. 44, n. 10, p. 951–976, 2017.

YEH, M. K.-C.; GOPSTEIN, D.; YAN, Y.; ZHUANG, Y. Detecting and comparing brain activity in short program comprehension using eeg. In: IEEE. **2017 IEEE Frontiers in Education Conference (FIE)**. [S. l.], 2017. p. 1–5.

## **APPENDIX A – DATASET DESCRIPTION**

The tables below describe the dataset created in this study, providing details and descriptions of its entities and their respective fields.

Table 9 – Jira Dataset

<b>Field</b>	<b>Description</b>
Key	A unique identifier for the issue
Type	Issue type. Can be: bug, improvement, new feature, task, sub-task, test or wish
Priority	The importance of the issue in relation to other issues
Status	The stage the issue is currently at in its lifecycle Can be: open, in progress, reopened, resolved or closed
Reporter	The person who entered the issue into the system
Assignee	The person to whom the issue is currently assigned
Components	Project component(s) to which this issue relates
Resolution	A record of the issue's resolution, if the issue has been resolved or closed. Can be: done, won't do, duplicate or cannot reproduce
InwardIssueLinks	List of issues that affect the current issue and their respective descriptions of how they affect it
OutwardIssueLinks	List of issues that are affected by the current issue and their respective descriptions of how they are affected
NoComments	How many developer comments there are in the issue
NoWatchers	How many people are watching the issue
NoAttachments	How many attachments there are in the issue
NoAttachedPatches	How many patch attachments there are in the issue
Summary	Issue title
SummaryTopWords	Preprocessing in the Summary field with the most frequent words and their respective frequencies
Description	A detailed description of the issue
DescriptionTopWords	Preprocessing in the Description field with the most frequent words and their respective frequencies
CommentsTopWords	Preprocessing in the Developers Comments with the most frequent words and their respective frequencies
CreationDate	The time and date on which the issue was entered into Jira
ResolutionDate	The time and date on which the issue was resolved
LastUpdateDate	The date the last issue update
FirstCommentDate	The date of the first developer issue comment
LastCommentDate	The date of the last developer issue comment
FirstAttachmentDate	The date of the first issue attachment
LastAttachmentDate	The date of the last issue attachment
FirstAttachedPatchDate	The date of the first issue patch attachment
LastAttachedPatchDate	The date of the last issue patch attachment
AffectsVersions	Project version(s) for which the issue is (or was) manifesting
FixVersions	Project version(s) in which the issue was (or will be) fixed

Table 10 – Issue Developer Comments Dataset

<b>Field</b>	<b>Description</b>
Key	A unique identifier for the issue
Author	The developer who made the comment on the issue
CreationDate	The date on which the comment was made
Content	The content of the comment
ContentTopWords	Preprocessing in the Content field with the most frequent words and their respective frequencies

Table 11 – Issue Changelog Dataset

<b>Field</b>	<b>Description</b>
Key	A unique identifier for the issue
Author	The developer who made the change on the issue
ChangeDate	The date on which the change was made on the issue
Field	The issue field that was modified
From	The content of the field before the change
To	The content of the field after the change

Table 12 – Issues-Commits Dataset

<b>Field</b>	<b>Description</b>
Issue Key	A unique identifier for the issue
Related Commit	The commit hash that is related to the issue
Previous Commit	The previous hash commit to the Related Commit

Table 13 – Commit Log Dataset

<b>Field</b>	<b>Description</b>
CommitHash	A unique identifier for the commit
Author	The person who originally wrote the work
AuthorDate	The datetime when the work was originally written
Committer	The person who last applied the work
CommitterDate	The datetime when the work was last applied
CommitMessage	The message of the commit
CommitMessageTopWords	Preprocessing in the Commit Message field with the most frequent words and their respective frequencies
FileName	The name of the file that was modified in the commit
FilePath	The path of the file that was modified in the commit
ChangeType	Type of the change made in the file. Can be: Added, Deleted, Modified, or Renamed.
IsSrcFile	Returns True if the modified file is a source code file, False otherwise
IsTestFile	Returns True if the modified file is a source code test file, False otherwise
ModificationAddLines	How many lines were added in the modified file
ModificationDelLines	How many lines were deleted in the modified file
CommitInsertions	How many lines were added in the commit
CommitDeletions	How many lines were deleted in the commit
Diff	Code changes in the commit
DiffAddedLines	Code added in the commit
DiffDeletedLines	Code deleted in the commit
NoMethods	Number of methods in the file after the change Is empty if the file is not a source code file
LoC	Lines Of Code (LOC) of the file
CyC	Cyclomatic Complexity of the file
NoTokens	Number of Tokens of the file

Table 14 – Git Dataset

<b>Field</b>	<b>Description</b>
Key	Issue key to which the commits are linked
CommitsMessageTopWords	Preprocessing in the Issue Linked Commits Message field with the most frequent words and their respective frequencies
HasMergeCommit	Returns True if there is at least one linked merge commit, False otherwise
NoCommits	How many commits have been linked to the issue
NoAuthors	How many authors are involved in the commits linked to the issue
NoCommitters	How many committers are involved in the commits linked to the issue
AuthorsFirstCommitDate	The authored datetime of the first commit that is linked to the issue
AuthorsLastCommitDate	The authored datetime of the last commit that is linked to the issue
CommittersFirstCommitDate	The committed datetime of the first commit that is linked to the issue
CommittersLastCommitDate	The committed datetime of the last commit that is linked to the issue
NonSrcAddFiles	How many non-source files were added in the commits linked to the issue
NonSrcCopyFiles	How many non-source files were copied in the commits linked to the issue
NonSrcDeleteFiles	How many non-source files were deleted in the commits linked to the issue
NonSrcModifyFiles	How many non-source files were modified in the commits linked to the issue
NonSrcRenameFiles	How many non-source files were renamed in the commits linked to the issue

**Table 14 continued from previous page**

<b>Field</b>	<b>Description</b>
NonSrcUnknownFiles	How many unknown non-source files were changed in the commits linked to the issue
NonSrcAddLines	How many lines were added to non-source files in the commits linked to the issue
NonSrcDelLines	How many lines were deleted to non-source files in the commits linked to the issue
SrcAddFiles	How many source files were added in the commits linked to the issue
SrcCopyFiles	How many source files were copied in the commits linked to the issue
SrcDeleteFiles	How many source files were deleted in the commits linked to the issue
SrcModifyFiles	How many source files were modified in the commits linked to the issue
SrcRenameFiles	How many source files were renamed in the commits linked to the issue
SrcUnknownFiles	How many unknown source files were changed in the commits linked to the issue
SrcAddLines	How many lines were added to source files in the commits linked to the issue
SrcDelLines	How many lines were deleted to source files in the commits linked to the issue
TestAddFiles	How many test files were added in the commits linked to the issue
TestCopyFiles	How many test files were copied in the commits linked to the issue
TestDeleteFiles	How many test files were deleted in the commits linked to the issue

**Table 14 continued from previous page**

<b>Field</b>	<b>Description</b>
TestModifyFiles	How many test files were modified in the commits linked to the issue
TestRenameFiles	How many test files were renamed in the commits linked to the issue
TestUnknownFiles	How many unknown test files were changed in the commits linked to the issue
TestAddLines	How many lines were added to test files in the commits linked to the issue
TestDelLines	How many lines were deleted to test files in the commits linked to the issue



Table 15 – BOHR Detailed Dataset

<b>Field</b>	<b>Description</b>
Class	The name of the class in which the AC was found
Atom	The type of AC
Snippet	The code snippet that contains the AC
Line	The line number where the AC was found

Table 16 – BOHR Aggregated Dataset

<b>Field</b>	<b>Description</b>
Version	The package name that contains Java classes
Classes	The number of Java classes contained in the package
LoC	The sum of lines of code of the Java classes contained in the package
Ocurrences	The number of ACs found in the Java classes contained in the package
AC Types	The number of AC Types found in the Java classes contained in the package
Classes with AC	The number of Java package classes that contain at least one AC
IOP	The number of Infix Operator Precedence Atoms found in the Java package classes
PreIncDec	The number of Pre-Increment/Decrement Atoms found in the Java package classes
PostIncDec	The number of Post-Increment/Decrement Atoms found in the Java package classes
CO	The number of Conditional Operator Atoms found in the Java package classes
OCB	The number of Omitted Curly Braces Atoms found in the Java package classes
LaCF	The number of Logic as Control Flow Atoms found in the Java package classes
AaL	The number of Arithmetic as Logic Atoms found in the Java package classes
CoLE	The number of Change of Literal Encoding Atoms found in the Java package classes
TC	The number of Type Conversion Atoms found in the Java package classes
RV	The number of Repurposed Variables Atoms found in the Java package classes

Table 17 – Changed/Deleted ACs Dataset

<b>Field</b>	<b>Description</b>
Key	A unique identifier for the issue
CommitHash	A unique identifier for the commit
CommitDeletions	How many lines were deleted in the commit
CommitInsertions	How many lines were added in the commit
Class	The name of the class in which the AC was found
Line	The line number where the AC was found
Atom	The type of AC
Snippet	The code snippet that contains the AC
Type	Issue type. Can be: bug, improvement, new feature, task, sub-task, test and wish
Action	Impact of the commit on the atom of confusion. Can be changed if the commit modifies the same line as the AC, preserving it; or deleted if the commit removes the AC

Table 18 – Commit ACs Changes Dataset

<b>Field</b>	<b>Description</b>
Issue	A unique identifier for the issue
Type	Issue type. Can be: bug, improvement, new feature, task, sub-task, test or wish
Related Commit	The commit hash that is related to the issue
Previous Commit	The previous hash commit to the Related Commit
PClasses	The number of Java classes before the changes of the related commit
PLoC	The sum of lines of code of the Java classes before the changes of the related commit
POcurrences	The number of ACs found in the Java classes before the changes of the related commit
PAcTypes	The number of AC Types found in the Java classes before the changes of the related commit
PClassesWithAC	The number of Java classes that contain at least one AC before the changes of the related commit
PIOP	The number of Infix Operator Precedence Atoms found in the Java classes before the changes of the related commit
PPreIncDec	The number of Pre-Increment/Decrement Atoms found in the Java classes before the changes of the related commit
PPostIncDec	The number of Post-Increment/Decrement Atoms found in the Java classes before the changes of the related commit
PCO	The number of Conditional Operator Atoms found in the Java classes before the changes of the related commit
POCB	The number of Omitted Curly Braces Atoms found in the Java classes before the changes of the related commit
PLaCF	The number of Logic as Control Flow Atoms found in the Java classes before the changes of the related commit
PAaL	The number of Arithmetic as Logic Atoms found in the Java classes before the changes of the related commit

**Table 18 continued from previous page**

<b>Field</b>	<b>Description</b>
PCoLe	The number of Change of Literal Encoding Atoms found in the Java classes before the changes of the related commit
PTC	The number of Type Conversion Atoms found in the Java classes before the changes of the related commit
PRV	The number of Repurposed Variables Atoms found in the Java classes before the changes of the related commit
RClasses	The number of Java classes after the changes of the related commit
RLoC	The sum of lines of code of the Java classes after the changes of the related commit
ROccurrences	The number of ACs found in the Java classes after the changes of the related commit
RAcTypes	The number of AC Types found in the Java classes after the changes of the related commit
RClassesWithAC	The number of Java classes that contain at least one AC after the changes of the related commit
RIOP	The number of Infix Operator Precedence Atoms found in the Java classes after the changes of the related commit
RPreIncDec	The number of Pre-Increment/Decrement Atoms found in the Java classes after the changes of the related commit
RPostIncDec	The number of Post-Increment/Decrement Atoms found in the Java classes after the changes of the related commit
RCO	The number of Conditional Operator Atoms found in the Java classes after the changes of the related commit
ROCB	The number of Omitted Curly Braces Atoms found in the Java classes after the changes of the related commit
RLaCF	The number of Logic as Control Flow Atoms found in the Java classes after the changes of the related commit
RAaL	The number of Arithmetic as Logic Atoms found in the Java classes after the changes of the related commit

**Table 18 continued from previous page**

<b>Field</b>	<b>Description</b>
RCoLe	The number of Change of Literal Encoding Atoms found in the Java classes after the changes of the related commit
RTC	The number of Type Conversion Atoms found in the Java classes after the changes of the related commit
RRV	The number of Repurposed Variables Atoms found in the Java classes after the changes of the related commit