Prevalence of 'Atoms of Confusion' in Open Source Java Systems: An Empirical Study

Nafis Fuad
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
bsse0920@iit.du.ac.bd

Noshin Tahsin
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
bsse0914@iit.du.ac.bd

Abdus Satter
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
abdus.satter@iit.du.ac.bd

***Abstract*—Atoms of confusion, or simply "atoms," are pieces of code that lead to misunderstanding while being interpreted. Previous research has shown that the presence of atoms has an effect on code readability. Aside from simple misunderstanding in lab setting, atoms of confusion are common and meaningful in open source C and C++ projects, and are thus removed by bug-fix commits. However, due to syntactical differences between language paradigms, the prevalence of atoms may vary in projects written in other languages (e.g. Java), which is yet to be explored. In this study, the first step is taken towards investigating the prevalence of 12 different atoms in the 13 most popular open-source Java projects. The relationship between the presence of atoms and aspects of code maintainability is also studied. Results show that, atoms are 4.7 time more prevalent in Java projects compared to open source C/C++ projects based on occurrence per line. For a total of 1085223 atoms in our corpus, they occur once every 4.8 lines. Some atoms are very obscure (e.g. the Logic As Control Flow atom which occurs once in 440060 lines). Some atoms are frequently occurring (e.g. the Infix Operator Precedence atom which occurs once in 6.4 lines). Impact of the presence of atoms on code maintainability is also explored. Besides, correlation between atoms are investigated. Results indicate that object oriented metrics contribute less in atom prevalence, whereas fine grained code-metrics have relatively better association.**

***Index Terms*—refactoring, software maintainability, software complexity**

I. Introduction

Object Oriented Programming was introduced to promote designing better human-understandable code. This change in paradigm came into play as systems built using structured programming demanded much cognitive effort to understand. However, despite design-level improvements, it is not free from code-level complexity. These confusing parts of the code are termed 'atoms of confusion' in literature.

When interpreting such confusing pieces of code, developers may not always reach the correct conclusions. They may misinterpret the meaning of code and misjudge the behavior of the program, leading to errors. This imposes a major threat to organizing long-term software development, maintenance, and evolution [1] [2]. To avoid this scenario, code should be written in such a way that developers can effectively understand and interpret it.

The term was originally introduced by Gopstein et al. [3] to detect portions of code in C programs that confuse developers. Existing literature mainly focused on the effects of these atoms in C code. The results show that when atoms when present in code, developers make significantly more mistakes [3]. An empirical study was done on open-source C projects, showing that developers are 2.7 to 56 times more likely to make errors in confusing code snippets [4]. More recent work by Langhout [5] derived such atoms of confusion relevant to Java code. However, to the best of our knowledge, no work has been done to empirically study the prevalence of such atoms in open-source Java systems. Besides, none of the studies explored the relationship between atoms and software maintainability.

In this study, the first step is taken towards exploring how the existence of atoms affects software maintainability in open-source Java projects. 13 most popular open source Java projects are selected for this purpose, namely Bukkit, Selenium, Commons-lang, Guava, Tomcat, AcionBarSherlock, Clojure, Spring Framework, Hadoop, Mockito, Facebook-Android-SDK, Netty, and ElasticSearch. 12 Java language specific atoms are considered in this study, which are Infix Operator Precedence, Post Increment Decrement, Pre Increment Decrement, Constant Variable, Conditional Operator, Arithmetic as Logic, Logic as Control Flow, Repurposed Variable, Dead, Unreachable, Repeated, Change of Literal Encoding, Omitted Curly Braces and Type Conversion. The existence of these atoms are detected in the selected projects through our regex-based detection approach. The maintainability metrics for the projects are calculated using the Understand tool. Then the correlation between atom types and code metrics is analyzed.

Analyzing the data, it is found that, atoms occur once every 4.87 lines, for a total of 1085223 atoms in our corpus. Some atoms are very obscure in Java projects e.g. the Repurposed Variable atom, while some are frequent e.g. the Infix Operator Precedence atom. The correlation between maintainability metrics and presence of atoms is presented and explained. For example, a strong correlation has been found between the Omitted Curly Braces atom and the three metrics - Number of Assignment Expressions, Number of Parenthesized Expressions, and Number of Comparisons.

The later sections of the paper are organized as follows. Section II contains related works, Section III contains background information, and Section IV describes the methodology. Sec-

TABLE I
ATOMS OF CONFUSIONS CONSIDERED IN THE STUDY

Atom Name	Java Code Snippet with Atom of Confusion	Java Code Snippet Free of the Confusion
Type Conversion (TC)	V1 = (int) 1.99f;	V1 = (int) Math.floor(1.99f);
Post-Increment/Decrement (POST-I/D)	V1 = V2++;	V1 = V2; V2 += 1;
Pre-Increment/Decrement (PRE-I/D)	V1 = ++V2;	V2 += 1; V1 = V2;
Constant Variables (CV)	V1 = V2;	V1 = 5;
Conditional Operator (CO)	V2 = V1 == 3 ? 2 : 1;	if (V1 == 3) { V2 = 2; } else { V2 = 1; }
Arithmetic as Logic (AAL)	(V1 - 3) * (V2 - 4) != 0	V1 != 3 && V2 != 4
Logic as Control Flow (LACF)	V1 == ++V1>0 —— ++V2 >0;	if (!(V1 + 1 >0)) { V2 += 1;} V1 += 1
Repurposed Variables (RV)	for(int V1 = 0;...; V1++) { for(int V2 = 0;...; V1++) {	for (int V1 = 0;...; V1++) { for (int V2 = 0;...; V2++) {
Dead, Unreachable, Repeated (DR)	V1 = 1; V1 = 2;	V1 = 2;
Omitted Curly Braces (OCB)	if (V1) F1(); F2();	if (V1) { F1(); } F2();
Change of Literal Encoding (CLE)	V1 = 013	V1 = Integer.parseInt("13", 8)
Infix Operator Precedence (IOP)	2 - 4 / 2	2 - (4 / 2)

tion V presents the evaluation and Section VI presents the threats to validity. Section VII concludes with future works.

II. Related Works

The concept of "atoms of confusion" in source code is relatively new in the research community and a few research efforts have been done regarding this concern. In the introductory work by Gopstein et al. [3], they derived 19 atoms of confusion in C code. Experimentation involving 73 subjects (university students with at least 3 months of C/C++ experience) was done to validate the code snippets hypothesized as confusing. The subjects were presented with similar code snippets, one containing an atom and the other transformed to remove the corresponding atom. Statistical significance was tested based on the evaluation task of interpreting the output of those code snippets.

Follow-up work from Gopstein et al. [4] shows that the 15 atoms that were proven statistically significant occur in practice once every 23 lines in open-source C/C++ projects. They demonstrated the real-world relevance of these selected atoms of confusion along with their rate of occurrence. Their study is based on an examination of 14 of the most well-known and influential C and C++ software projects. Their study also shows that atoms tend to be removed more in case of bug-fixing changes rather than other types of changes. This indicates the relationship of atoms causing confusion and confusion causing misinterpretation leading to bugs. Numerically, bug-fixing commits are 1.25 times more likely to remove atoms than non-bug fixing commits. This extensive study also shows that in open-source systems atoms are 1.13 times more commented than non-atom code. Medeiros et al. [6] also investigated the frequency of occurrences of most of the atoms and discovered that all expect one occur in the projects studied. They used a mixed method approach, including repository mining and developer surveys, to collect data from 50 open-source C projects. In a similar fashion with previous work [3], participants perceived the version affected by the atom of confusion to be more confusing and/or less readable when presented with both versions of the code snippets.

Langhout et al. [5] derived atoms relevant to Java code and found that participants are 2.7 to 56 times more likely to make errors in code snippets with atoms of confusion. Participants perceived the version affected by the atom of confusion to be more confusing and/or less readable when presented with both versions of the code snippets. Atoms specific to C/C++ syntax were avoided in this study. However, no new atom specific to Java syntax or Object Oriented paradigm were introduced. Bogachenkova et al. [7] looked into the possible cause-and-effect relationship between atoms of confusion and confusion in code reviews. They also investigated how those atoms of confusion evolve over the course of pull requests.

Although work has been done to identify atoms of confusion for different languages, to the best of obtained knowledge, none of the studies demonstrate the evolution of software containing atoms. There has been no direct investigation into the relationship between atoms of confusion and code maintainability.

III. Background

Based on the origin source, 'confusion' is defined as - "what happens when a person and a machine read the same piece of code, yet come to different conclusions about its output". Atoms of confusion are the smallest possible patterns of misinterpretable source code. The presence of atoms makes the code more confusing and leads to more mistakes. Gopstein et al. [3] first observed these patterns in source code. They noticed that failure in notable software bug examples is caused by "single, well-contained, programming errors as at the syntactic or semantic level, rather than the algorithmic or system-levels of the project". They defined these minimal portions of code as "atoms of confusion," or "atoms". Castor [8] expanded this definition by formalizing atoms as precisely identifiable and likely to cause confusion, which is replaceable by a functionally equivalent code pattern that is indivisible and less likely to cause confusion.

In this study, 12 types of atom specific to the Java Language is considered which are Infix Operator Precedence, Post Increment Decrement, Pre Increment Decrement, Constant Variable, Conditional Operator, Arithmetic as Logic, Logic

as Control Flow, Repurposed Variable, Dead, Unreachable, Repeated, Change of Literal Encoding, Omitted Curly Braces and Type Conversion. The atoms along with their respective 'free of confusion' counter parts are presented in Table I.

CK metrics [9] are used as code level metrics in this study. The metrics are Coupling Between Objects (CBO), Depth Inheritance Tree (DIT), Response for a Class (RFC), Weight Method Class (WMC), Lines of code (LOC), Lack of Cohesion of Methods (LCOM), Number of Static Invocations (NOSI), Number of Methods (nMethod), Number of Comparison Expression (nComparison), Number of Loops (nLoop), Number of Parethesized Expression (nParenthesizedExps), Number of Assignments (nAssignments) and Number of Static Fields (nStaticFields).

IV. Methodology

This study aims to investigate how the presence of atoms in code is related with software maintainability. To achieve this goal, the presence of atoms in the subject systems is detected through an regex-based approach. The maintainability metrics for the subject systems are calculated using the Understand tool. Finally, the relationship between atoms and maintainability metrics is explored. An overview of the steps in this empirical study can be depicted from Fig. 1.

A. Detecting Atoms

In this study, atoms are detected using a regex-based approach. As different atoms have different syntax, the logic for detecting each of the distinct atom was also different. The detection methods for each of the 12 atoms considered in this study are described in detail below.

1) ***Infix Operator Precedence****:* To check the existence of this atom, the total number of occurrences of mathematical operators (+, -, *, /) in the statement was counted. This number is referred to as (*numop*). The total number of occurrences of parenthesis pairs ('()') is also calculated. This number will be referred as *numpar*. Now, if *numop* is even and *numpar* is less than *numop*/2 in any statement, the statement is detected as the infix operator precedence atom. Again, if *numop* is odd and *numpar* is less than *numop*/2 + 1 in any statement, the statement is also detected as the infix operator precedence atom.

For example, in the statements *result = (2/4)-(2*3)-9* and *result = (2/4)-(2*3)-(9/3)*, atom *Infix Operator Precedence* is not present. Whereas, in *result = (2/4)-(2*3)-9+3*, atom *Infix Operator Precedence* is present.

2) ***Post Increment Decrement****:* A statement is taken and white spaces are removed. The existence of '=' is checked in the statement. If the statement contains a '=', does not contain any '==' and ends with '++', it is split considering '=' as a separator. Following the split, everything on the left hand side of the = is referred to as LHS. Similarly, everything on the right side of the = sign is referred to as RHS. The '++' is removed from the RHS and the new RHS (RHS') is compared with the LHS. If the LHS and RHS' does not match, it can be determined that the statement contains the Post Increment Decrement atom. For example, the statement*a = b++* is such an atom, while a = a++ is not.

3) ***Pre Increment Decrement****:* A statement is taken and white spaces are removed. The existence of '=' is checked in the statement. If the statement contains a '=', does not contain any '==' and ends with '++', it is split considering '=' as a separator. Everything on the left hand side of the = after splitting is referred to as LHS. Likewise, everything on the right side of the = is referred to as RHS. The '++' is removed from the RHS and the new RHS (RHS') is compared wih the LHS. If the LHS and RHS' does not match, it can be determined that the statement contains the Pre Increment Decrement atom. For example, the statement*a = ++b* is such an atom, while a = ++a is not.

4) ***Constant Variable****:* A statement is taken, white spaces are removed from it and the existence of '=' is checked in the statement. If '=' is found in the statement, it is checked whether any kind of arithmetic operation is taking place there. This must be checked because if an arithmetic operation occurs, such as *V1 = V2 + 3*, the statement will not fall under the Constant Variable atom. If no mathematical operators (+, -, *, /) are found in the string, it is possible to conclude that no arithmetic operation is taking place in that statement. Then the statement is split considering '=' as a separator. The RHS is taken and regex is used to determine whether it is a variable name or a literal. If the RHS is recognized as a variable name, the statement is identified as having the Constant Variable atom.

For example, in the statement *V1 = V2*, the RHS will be V2, which is a variable name. As a result, the statement is classified as a Constant Variable atom. In case of the statements *V1 = 5* and *V1 = "some string"*, the RHS is *5* and *"some string"*, respectively. Both of the RHSs are literals. Therefore, the statements does not fall under the category of the Constant Variable atom.

5) ***Conditional Operator****:* To identify the existence of this atom, regex is used. The presence of both '?' and ':' is identified in the statement first. If found, the statement is considered as an Conditional Operator atom.

V2 = V1 == 3 ? 2 : 1; is an example of Conditional Operator atom, as both '?' and ':' are present here. Whereas if *(V1 == 3) V2 = 2; else V2 = 1;* is the confusion free version of this statement.

6) ***Arithmetic as Logic****:* To find this type of atom, the presence of both mathematical and relational operators in the same statement is looked for. If found, the statement is considered to contain the atom in concern.

For example, *(V1 - 3) * (V2 - 4) != 0* is an example of arithmetic as logic atom. This contains both mathematical and relational operators. On the other hand, *if (V1 != 3) (V2 != 4)* is the confusion free version of it as it does not contain mathematical and relational operators together in the same statement.

7) ***Logic as Control Flow****:* To detect these kind of atoms, the presence of both mathematical and logical operators in the same statement is looked for. If found, the statement is

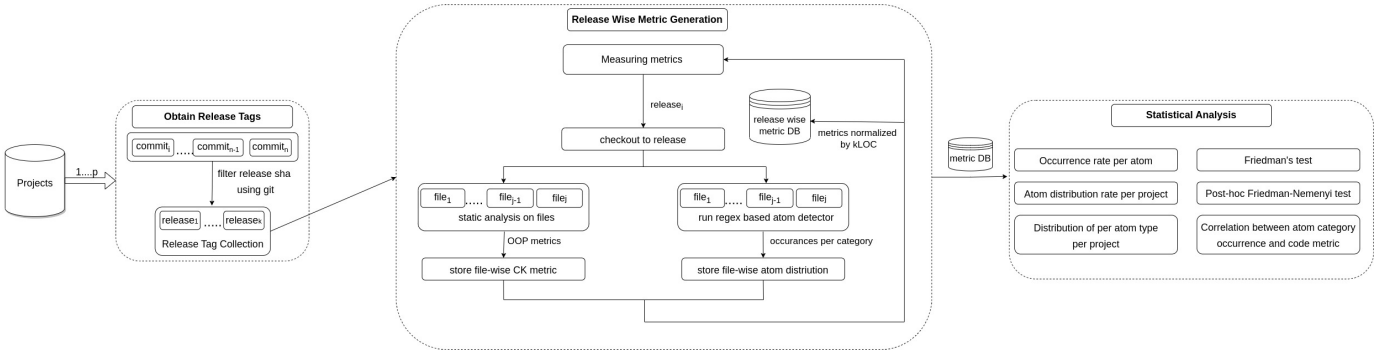

Fig. 1. Study Overview

considered to contain the atom in concern. *V1 == ++V1 > 0 || ++V2 > 0;* is an example of this kind of atom as it contains both mathematical and logical operators in it.

8) ***Repurposed Variable****:* To detect this type of atom, search was conducted first to look for nested for loops. Then, it was checked whether the iterator initialized in the first loop control statement, instead of its respective initialized iterator, is incremented in the second loop control statement. If yes, then the existence of a Repurposed Variable atom is confirmed. An example of such kind of atom and its confusion free version can be found in Table I.

9) ***Dead, Unreachable, Repeated****:* To detect this atom, pairs of consecutive statements are taken first. Each of the statements in the pair is split considering = as separator. The left hand sides of the = in both of the statements are compared. If they match, the existence of Dead, Unreachable, Repeated atom is confirmed.

For example, for the consecutive statement pair (*'V1 = 1;', 'V1 = 2';*), the left hand sides after splitting them both considering = as separator are 'V1' and 'V1', respectively.Therefore, this snippet is considered as the Dead, Unreachable, Repeated atom.

10) ***Change of Literal Encoding****:* To detect atom of this category, each statement was taken and the white spaces were removed. For example, the statement *V1 = 013* was converted to *V1=013*. Then it was checked whether their is any = in the statement. If any = exist, the statement is split considering '=' as the separator. The right hand side of the statement is taken. Using regex, it is checked that whether the right hand side starts with a 0 and whether its length is more than 1. The length of the RHS indicates whether it is an atom or simply that a value assignment is taking place there. If both the condition applies, that is, if such a statement was found where the term right to an '=' having a length greater than 1 starts with a 0, that statement was identified as an Change of Literal Encoding atom.

For example, in case of the statement *V1 = 013*, 013 is the right hand side when split considering '=' as the separator. 013 starts with a 0 and has a length greater than 1. Therefore, the statement *V1 = 013* contains the Change of Literal Encoding atom. On the other hand, the statement V1 = 0 is not an atom. Though 0 starts with a 0, it does not have a length greater than 1. Therefore, the statement *V1 = 0* does not contain the Change of Literal Encoding atom. *System.out.println (" The output is V1 = 013");* is another example that contains the Change of Literal Encoding atom.

11) ***Omitted Curly Braces****:* A statement is taken and existence of one of the keywords 'if', 'while', 'for' is checked in that statement. If found, the existence of curly brace ('{') is checked next. If no curly brace is found, the statement can be concluded to contain the Omitted Curly Brace atom.

For example, the statement *if (v1) f1(); f2();* contains an if block. However, as there are no curly braces, it may become confusing for the developer whether *f2()* falls under the if block. Therefore, this is an example of the Omitted Curly Braces atom. On the other hand, the statement *if (v1) f1(); f2();* clearly separates the if block from the consecutive portion *f2()* and shows that *f2()* does not fall under the if block. Therefore, this is a confusion free version of the previous example containing the Omitted Curly Braces atom.

12) ***Type Conversion****:* To detect this atom, the existence of '(int)'/'(double)'/'(float)' is checked in a statement. If one of these keywords exists, it is checked if 'Math.floor' or 'Math.ceil' is present in that statement. If not found, the statement is determined to be containing the Type Conversion atom.

For example, in the statement *v1 = (int) 1.99f;*, the presence of keyword *(int)* indicates that a type conversion is taking place here. However, as there is no 'Math.floor' or 'Math.ceil' here, this may lead to confusion. Therefore, this is an example of the Type Conversion atom. On the other hand, *v1 = (int) Math.ceil(1.99f);*, *v1 = (int) Math.floor(1.99f);* are examples of the confusion free version of the Type Conversion atom. Type conversion is taking place in these statements and the presence of 'Math.floor' and 'Math.ceil' does not leave any confusion for the developers regarding the actual output.

B. Generating Code Metrics

Calculation of code level metric is necessary to analyse relationship of code and atom evolution. CK metrics are selected as they include the most popular metrics to evaluate class-level Object Oriented Systems [9]. Release tag information obtained from each project's repository is used to checkout to each release state. Afterwards, file/class wise code metrics

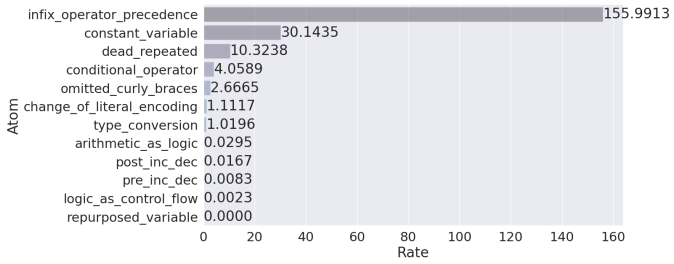

Fig. 2. The rate of individual atoms per KLOC across all projects

are generated for each release. Adding up metric values for the files, the release level metric score is calculated. A dataset of CK code metrics per release is generated for each of the subject systems.

C. Determining Metric Relationship

Similar to code metric generation, atom occurrences of each revision is calculated per file. Afterwards, the file-level occurrences are added to get project-level atom occurrences per revision. Both code metrics and atom counts were normalized using the project size in kLOC to tackle any confounding effect due to size. The two evolution data is used to find correlation between types of atoms and code metrics.

V. Evaluation

A. Subject Selection

The subject systems written in Java were selected based on their popularity on GitHub, following previous work [10]. Thirteen projects were ranked using GitHub Stars and number of contributors from the GHTorrent dataset[1], namely Bukkit, Selenium, Commons-lang, Guava, Tomcat, AcionBarSherlock, Clojure, Spring Framework, Hadoop, Mockito, Facebook-Android-SDK, Netty, and ElasticSearch. The selected projects have maturity of more than 10 years and 226.5 releases on average. Furthermore, the systems being open-source, the coding style is not biased towards any individual contributor. The release tags obtained from the repositories are used to obtain release-wise file level atom distribution as well as code metric. The file-level metrics are merged to obtain the release-level metric distribution, which acts as the main dataset for analysis.

B. Prevalence of atoms

1) ***How frequently do atoms occur in Java projects?****:* According to our findings, atoms of confusion appear more frequently in Java projects. Atoms appeared at a rate of 20.53% in our corpus of 13 large and significant open-source Java projects, implying that one in every 4.87 lines in our corpus contains an atom of confusion. In total, over 1085200 atoms of confusion is discovered, with up to 349617 and 256491 per project in larger code bases like Hadoop and Elasticsearch respectively.

By definition, each type of confusion atom has its own syntax and semantics. The method of causing misunderstanding in the readers also varies depending on the type of atom. As a result, the prevalence of each pattern was examined separately. Figure 1 depicts the rates of occurrence of individual atoms per KLOC. It has been discovered that the most frequently occurring atom is the Infix Operator Precedence. This is because the software engineering community discourages the use of 'unnecessary' parentheses. Furthermore, IDEs and code formatting tools offer runtime suggestions to remove "unnecessary" parentheses. As a result, the Infix Operator Precedence atom prevails in the code, making the code confusing to developers. It is also found that the Constant Variable atom is the second most frequent atom occurring in open source Java projects. However, the reason behind this is unclear.

The least frequent atom of confusion is the repurposed variable atom. This is obvious because at the beginning of the loop condition. It is highly unlikely that they would initialize one iterator while incrementing or decrementing another. The logic as control flow atom is the second least common one. This is also very obvious, as developers tend to use this kind of complex syntax less often.

The key takeaway from our analysis is that, atoms are 4.7 times more prevalent in Java projects compared to C/C++ projects studies in literature [4]. They occur once every 4.87 lines, for a total of 1085223 atoms in our corpus. Some atoms are very obscure in Java projects e.g. the Logic As Control Flow atom that occurs once in 440060 lines. They can also be frequently occurring e.g. the Infix Operator Precedence atom that occurs once in 6.4 lines.

2) ***Do atoms occur at different rates in different projects?****:* From our analysis, it is seen that atoms occur at different rates in different projects. Figure 3 depicts the overall rate at which atoms appear in the projects chosen as the subject of this study. It is found that Bukkit has the most atoms proportional to its size, while Guava has nearly one-third of atoms compared to Bukkit.

Figure 4 shows the rate of individual atoms in each project. The rates are calculated by first identifying the project with the highest usage and then normalizing the data from each project against that maximum. Each column in the heatmap depicts an individual project compared to the entire corpus. The column for each project ranks the atoms based on lowest to highest occurrence. Each row depicts the atom usage from project to project.

3) ***Do some atoms occur consistently more than others?****:* Friedman's test[2] is employed to determine whether there is at least one atom always occurring more than any other atom. The application scenario of Friedman's test can be illustrated from the classical example - *"N wine judges each rate K different wines. Are any of the K wines ranked consistently higher or lower than the others?"*. The normalized occurrences per atom for all 13 projects are grouped to perform the Friedman's test. The null hypothesis is that the occurrence of

[1]http://ghtorrent.org/msr14.html

[2]https://en.wikipedia.org/wiki/Friedman_test

TABLE II
POSTHOC FRIEDMAN NEMENYI TEST FOR ALL ATOM CATEGORIES. BOLD VALUES INDICATING STATISTICAL SIGNIFICANCE USING P-VALUE OF 0.05 AND (*) INDICATING SIGNIFICANCE AFTER BONFERRONI CORRECTION

Atom Types	TC	POST-I/D	PRE-I/D	CV	CO	AAL	LACF	RV	DR	OCB	CLE	IOP
TC	1											
POST-I/D	0.643	1										
PRE-I/D	0.433	0.9	1									
CV	0.127	**0.001***	**0.001***	1								
CO	0.9	**0.027**	**0.008**	0.9	1							
AAL	0.626	0.9	0.9	**0.001***	**0.025**	1						
LACF	0.071	0.9	0.9	**0.001***	**0.001***	0.9	1					
RV	**0.032**	0.9	0.9	**0.001***	**0.001***	0.9	0.9	1				
DR	0.452	**0.001***	**0.001***	0.9	0.9	**0.001***	**0.001***	**0.001***	1			
OCB	0.9	0.136	0.055	0.626	0.9	0.127	**0.003***	**0.001***	0.9	1		
CLE	0.9	0.591	0.375	0.157	0.9	0.574	0.055	**0.025**	0.506	0.9	1	
IOP	**0.009**	**0.001***	**0.001***	0.9	0.452	**0.001***	**0.001***	**0.001***	0.9	0.146	**0.012**	1

TABLE III
CORRELATION BETWEEN ATOM TYPES AND CODE METRICS. BOLD VALUES INDICATING MODERATE OR STRONG ASSOCIATION

Code Metric/ Atom Type	CBO	DIT	RFC	WMC	LOC	LCOM	NOSI	nMethod	nCompa-rison	nLoop	nParenthe-sizedExps	nAssign-ments	nStatic-Fields
TC	-0.39	-0.30	-0.37	0.15	0.08	0.42	0.04	-0.18	0.25	0.47	0.27	0.19	0.08
POST-I/D	-0.36	-0.33	-0.24	0.23	-0.18	0.12	0.13	-0.35	0.42	0.40	0.43	0.36	0.43
PRE-I/D	-0.25	-0.23	-0.21	0.05	-0.00	0.08	0.29	-0.15	0.19	0.08	0.04	0.09	-0.05
CV	**-0.60**	**-0.61**	-0.42	0.51	0.05	0.09	-0.12	-0.45	**0.71**	0.48	0.52	0.57	**0.65**
CO	0.05	-0.07	-0.04	0.20	0.12	0.08	0.12	0.02	0.18	0.14	0.18	0.25	0.01
AAL	0.28	0.22	0.12	0.16	0.01	-0.21	-0.24	0.29	-0.11	0.18	-0.07	-0.18	-0.21
LACF	0.07	-0.01	0.09	0.11	-0.04	-0.05	0.02	-0.00	0.05	0.02	0.03	0.04	0.01
RV	0.05	-0.03	0.01	-0.01	-0.01	-0.09	-0.08	-0.07	0.08	0.19	0.06	0.00	-0.06
DR	-0.58	-0.58	-0.46	0.25	-0.03	0.31	**0.62**	-0.49	**0.69**	0.35	0.42	**0.68**	**0.63**
OCB	-0.36	-0.24	-0.44	0.56	-0.17	0.35	0.50	-0.24	**0.64**	0.42	**0.66**	**0.70**	0.20
CLE	-0.45	-0.26	-0.57	0.55	-0.09	0.31	0.08	0.06	0.34	0.35	0.49	0.25	0.27
IOP	-0.15	0.18	-0.22	0.39	-0.30	0.01	-0.03	0.46	-0.08	-0.31	-0.02	-0.26	-0.16

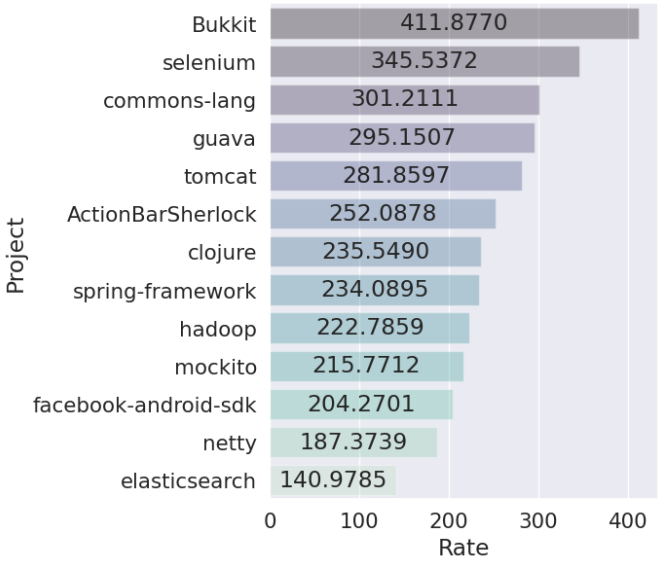

Fig. 3. The rate of atoms in each project

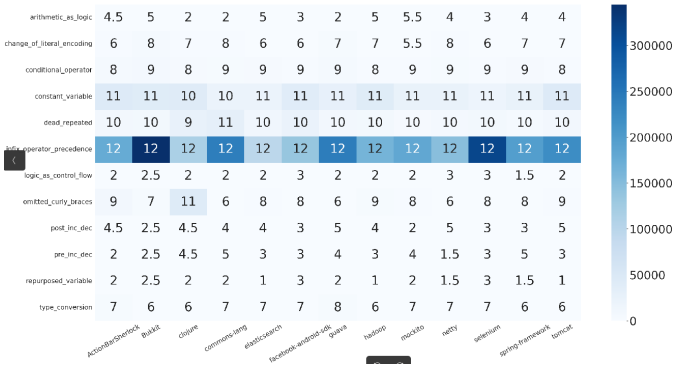

Fig. 4. The rate of individual atoms per LOC across all projects

each type of atom is not consistently different among projects. A test statistic=135.261 and p-value=1.49e-23 is yielded for this test. Therefore, rejecting the null hypothesis indicating statistically the existence of atoms consistently occurring more. Moreover, posthoc Friedman-Nemenyi test is employed to further evaluate every pair of categories. In this pair-wise test the null hypothesis is that the two categories occur similarly in open-source projects. The statistical significant is evaluated after applying the Bonferroni correction [11] to counteract the multiple comparisons problem[3]. In this process the target significance level of 0.05 is divided by 12 (number of statistical tests performed). The corrected level of significance is now 0.004167.

TABLE II shows the pair-wise statistical p-values of the posthoc Friedman-Nemenyi test. The bold cells indicate pairs

[3]https://en.wikipedia.org/wiki/Multiple_comparisons_problem

rejecting the null hypothesis using a p-value of 0.05 whereas '*' indicates the pairs that are significantly different even with Bonferroni corrected p-value. For example, the null hypothesis is rejected to confirm that *Constant Variable* always consistently occur more than *Post Increment/Decrement*, *Pre Increment/Decrement*, *Arithmatic As Logic*, *Logic As Control Flow*, and *Repurposed Variable*. Even if *Infix Operator Precedence* is seen to occur much more in open source (based on Fig. 2), it is also statistically validated from TABLE II. However, *Infix Operator Precedence* does not always consistently occur more than other types of atom.

C. Correlation with Maintainability

Previous studies have shown that, atoms have a negative impact on the readability of code. In this study, how the presence of atoms in code and the maintainability aspects of code correlate is explored. For this purpose, the prepared dataset containing the atom and code metric distribution per revision is used. The normalized values are used to determine the association of each code metric with different categories of atoms. TABLE III lists the correlation coefficient between each atom type and CK code metric. The atom types and code metric names are abbreviated due to space constraints. The descriptions can be obtained from TABLE I and public availability [9] respectively. The pairs of categories that have moderate to high correlation (positive or negative) are made bold in TABLE III. This table depicts that there is a strong correlation between *Omitted Curly Braces* with *Number of Assignment Expressions*, *Number of Parenthesized Expressions*, and *Number of Comparisons*. This finding is straightforward as developers often use one line conditional sentences to assign values to variables or call certain methods. Therefore the increase in the number of comparisons and assignment operations cause an increase in *Omitted Curly Braces*. *Constant Variable* atom is positively correlated with *Number of Comparisons* and *Number of Static Fields*. The intuition behind this relationship is that static fields are supposed to be constant intentionally and developers usually assign values based on comparison outcome. *Constant Variable* is negatively correlated with *Coupling Between Objects* and *Depth of Inheritance Tree*. This relationship can be hypothesised to hold as when coupling increases much of the responsibility if deferred to other classes, needing less constant variables. On the other hand, on increased inheritance depth, less variables need to be defined explicitly as inherited attributes are used frequently.

Initially proposed atoms of confusion was introduced to expose ambiguities in C/C++ projects, more generically, structured programming languages. Recently, atoms relevant to Java were proposed. However, Java being an Object Oriented language, contains code not only confusing on a structural level rather involving Object Oriented specific patterns for example, Polymorphism or Liskov Substitution Principal. TABLE III shows that atoms of confusion are more associated with structural metrics like *Number of Comparison, Assignment, etc.* rather than Object Oriented metrics such as complexity (*WMC*), cohesion(*LCOM*), etc. Confusing code is prevalent in open-source Java systems, however there is lack of proper definition of atoms in case of Object Oriented paradigm.

VI. Threats to Validity

1) Threats to External Validity: The presence of the twelve Java specific atoms initially proposed by Gopstein et al. [3] is investigated in this study. The 13 most popular open-source Java projects have been chosen as the subject of the study. However, there could be more Java-specific atoms of confusion that haven't been proposed or defined in the literature and thus are not covered by us.

2) Threats to Internal Validity: In this study, it is investigated whether the presence of atoms has any effect on code maintainability. The relationship between the presence of atoms and code maintainability has been investigated for this purpose. There may have been other factors involved that impacted the maintainability aspects of the code. We may not have been able to completely eliminate their influence.

3) Threats to Construct Validity: The detection method is carefully constructed based on existing literature. Twelve distinct types of atoms were identified based on their definition and distinct syntax. The Understand tool, which is widely used in the literature, was used to measure the maintainability metrics.

4) Threats to Content Validity: The content of our atom detection and maintainability measurement methods covers all relevant parts of the subject they aim to measure, i.e. twelve types of atoms and maintainability metrics, respectively. Irrelevant aspects were not included in the measurement method.

VII. Conclusion and Future Works

Atoms or atoms of confusion are the smallest possible patterns of source code that can be misinterpreted. Previous studies showed that, presence of these atoms affects the readability of code and makes the code more error-prone. However, those findings were based on C and C++ projects. In this study, the prevalence of atoms is investigated on open-source Java projects. This is the first step towards exploring atoms in Java projects. Different atoms are detected using relevant regex-based methods in this study. The relation between the presence of atoms and maintainability metrics are also explored and explained.

The future agenda includes investigating more Java-specific atoms of confusion that haven't been proposed or defined in the literature yet. Experimentation also need to be done taking both 'projects containing atoms' and 'projects without atoms' into consideration.

References

[1] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending studies on program comprehension," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 308–311.

[2] J. Siegmund, "Program comprehension: Past, present, and future," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 13–20.

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

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

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

[6] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, and R. Gheyi, "An investigation of misunderstanding code patterns in c open-source software projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1693–1726, 2019.

[7] V. Bogachenkova, L. Nguyen, F. Ebert, A. Serebrenik, and F. Castor, "Evaluating atoms of confusion in the context of code reviews."

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

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

[10] S. F. Huq, A. Z. Sadiq, and K. Sakib, "Is developer sentiment related to software bugs: An exploratory study on github commits," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 527–531.

[11] E. W. Weisstein, "Bonferroni correction," *https://mathworld. wolfram. com/*, 2004.