Reevaluating the Defect Proneness of Atoms of Confusion in Java Systems

Guoshuai Shi
University of Waterloo
Waterloo, Ontario, Canada
g7shi@uwaterloo.ca

Farshad Kazemi
University of Waterloo
Waterloo, Ontario, Canada
f2kazemi@uwaterloo.ca

Michael W. Godfrey
University of Waterloo
Waterloo, Ontario, Canada
migod@uwaterloo.ca

Shane McIntosh
University of Waterloo
Waterloo, Ontario, Canada
shane.mcintosh@uwaterloo.ca

ABSTRACT

Background: *Code confusion* concerns source code characteristics that make code harder for authors and reviewers to comprehend. *Atoms of Confusion* (AoCs) are a set of low-level programming idioms for C-like languages that have been proposed as a potential source of code confusion; previous studies have empirically evaluated the extent to which they (i) are confusing to developers and (ii) introduce risk to software products.

Aims: In this study, we further explore *Atoms of Confusion* and question the assumptions associating them with defects, and associating their removal with defect-fixing activities.

Method: We mine 76,610 pull requests from six Java open-source projects, extracting and analyzing changes relating to AoCs.

Results: First, we find no relation between the existence of AoCs and defect-fixing activity. Second, we observe that for some types of AoC—such as *infix operator precedence* and *conditional operator*—although quantitative analysis suggests a relation between their removal and fixes for defects, removing them does not contribute to the defect-fixing process. Finally, we find that project- and language-specific factors can affect the prevalence of AoC types, such as *pre-increment/decrement* and *type conversion* AoCs.

Conclusion: While prior work reported that AoCs impact defect proneness in C and C++ systems, we find that the presence of AoCs did *not* affect defect proneness in open-source Java projects. Our results suggest that future work is needed to investigate project- and language-specific factors such as project style guides and implicit type conversion that may impact the defect proneness of AoCs.

CCS CONCEPTS

• **Software and its engineering → Consistency**; *Maintaining software*; Object oriented development.

ESEM '24, October 24–25, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-1047-6/24/10…$15.00
https://doi.org/10.1145/3674805.3686677

KEYWORDS

program comprehension, Atoms of Confusion, defect proneness

ACM Reference Format:
Guoshuai Shi, Farshad Kazemi, Michael W. Godfrey, and Shane McIntosh. 2024. Reevaluating the Defect Proneness of Atoms of Confusion in Java Systems. In *Proceedings of the 18th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), October 24–25, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3674805.3686677

1 INTRODUCTION

As a multifaceted and dynamic process encompassing a variety of coordinated tasks—such as requirements gathering, design, coding, testing, and deployment [20]—software development is susceptible to confusion at any phase [9, 14]. Confusion can contribute to defects (a.k.a. bugs) [15, 28], project delays [16, 19], diminished code review quality [10, 11], and other negative consequences [12, 33].

Prior work has explored different forms of confusion that can occur during software development as well as their impact. Several knowledge-oriented and technical factors that contribute to confusion have been identified, such as a lack of familiarity with the codebase, a lack of programming skills [10, 18], unclear code patterns [14, 22], and poor code readability [7, 28]. Furthermore, studies have shown that human factors, such as cultural and communication differences among team members, can cause problems during development [17]; natural-language barriers and differing interpretations of project requirements have also been shown to contribute to developer confusion [16].

While research has shown that confusion does impede software development [7, 10, 15], precisely identifying *when* a developer is experiencing confusion can be challenging. To this end, several methods to identify and study developer confusion have been proposed. For example, Ebert *et al.* [9] inspected 800 code review comments and built a tool to detect when instances of confusion occur. Gopstein *et al.* [14] proposed *Atoms of Confusion* (AoCs)—a set of 15 low-level programming idioms endemic to C-like languages that are believed to confuse readers of the code. In their study, they surveyed 73 students and found that these idioms were indeed confusing, although to different extents. In follow-up work, Gopstein *et al.* [15] studied the prevalence and defect-inducing nature of AoCs in long-lived projects written in C and C++. Recently, Pinheiro *et al.* [27] discovered that AoCs that were removed in defect-fixing

and improvement commits were rarely the cause of the underlying issue that prompted the fix. Their inspection of 77 AoCs that were removed in defect-fixing and improvement commits revealed that only nine were implicated as inducing the changes.

While prior work has focused on the impact of the removal of AoCs, few studies have explored their addition. Bogachenkova *et al.* [4] was the first to consider the addition of AoC in *Pull Request*s (PRs). They hypothesized that AoC might not be the source of confusion among senior developers. To this end, they extracted AoCs from 12,100 closed PRs from the openHAB add-ons[1] and ksqlDB[2] repositories, and then manually identified comments expressing confusion. They found that no correlation existed between AoC and confusion in comments and that AoCs often persisted in the code after a PR. However, studying the confusion in comments, while valuable on a smaller scale [9], cannot be expanded easily, limiting the generalizability of such studies.

In this study, we address the limitations of previous studies [4, 15] and extend them by reevaluating whether AoCs contribute to defects in Java projects as a result of their confusing nature. To this end, we identify changes related to AoCs in PRs originating from six large, open-source Java projects. We then analyze the correlation of AoCs with the type of PR—distinguishing between *Bug-Fixing PR*s (BFPRs) and *Non-Bug-Fixing PR*s (NBFPRs). We structure our study by addressing the following Research Questions (RQs).

RQ1 Are there any associations between the trend of AoC changes and PR type?
Motivation: We strive to determine whether Java projects exhibit the same patterns identified in C++ projects by Gopstein *et al.* [15], which are associated with inducing defects, or align more closely with the findings of Bogachenkova *et al.* [4], which suggest that there is no significant correlation. To do so, we study whether the type of PR is correlated with a change in the incidences of AoCs.
Results: We find that in one project, bug-fixing PRs are associated with a significant increase in the number of AoCs when compared to non-bug-fixing PRs. However, in the other five projects, the results are inconclusive, which is consistent with the findings of Bogachenkova *et al.* [4].

RQ2 Do PR types correlate with relative AoC removal and addition rates?
Motivation: While Gopstein *et al.*'s study [15] controlled for the impact of commit size on AoC changes, they focused on AoC removal and ignored addition. We replicate their study using our dataset, investigating the association between the removal and addition of AoCs and the PR "type" (i.e., *Bug-Fixing PR* or *Non-Bug-Fixing PR*). We strive to determine if the Java projects are also impacted by AoCs similar to the C/C++ projects in the original study.
Results: AoCs are removed more often in BFPRs than in NBFPRs in the six studied projects; however, AoCs are also added more often. Follow-up statistical tests show that while such rates are meaningful in the C/C++ case, there is no conclusive difference between the PR types, and we cannot attribute defects to AoCs in our studied Java projects.

RQ3 Why are different types of AoC removed and added in bug-fixing and non-bug-fixing PRs?
Motivation: In prior work, the different types of AoC were not all found to be equally confusing [8, 14, 33]. Thus, we set out to investigate whether different types of AoC are treated distinctly in different types of PR, and whether some of them are more likely to be defect-inducing and thus more likely confusing to Java developers.
Results: From a statistical perspective, four of the ten AoC types seem to be defect-inducing; however, we find that removing them is not an explicit part of the implicated maintenance activity. Instead, we conjecture that AoCs are added by developers due to their perceived convenience or utility. We posit that project- and language-specific factors could impact the removal and addition of different types of AoC.

Our study challenges the previous hypothesis that AoCs are defect-inducing [15]. We conclude that since the removal and addition of AoCs varies across the studied Java projects, it is unclear whether AoCs are responsible for defects. While AoCs are removed more often in BFPRs, they are also added more often in BFPRs. Moreover, the removal of different types of AoC could be unintentional or dependent on project- or language-specific factors such as project style guides and implicit type conversion. Finally, we recommend that future studies control for these confounding factors that may contribute to whether AoCs are confusing, and investigate the perceptions of developers on AoCs in different contexts.

Data Availability. We prepare a replication package[3] to assist future research. We also provide an online appendix[4] which explains and explores related contexts, such as technical debt.

2 BACKGROUND AND RELATED WORK

While contributors may experience confusion during various stages of software development, confusion most commonly adversely affects development in two situations: (1) during code review discussions [10, 11], and (2) while reading code during development tasks, such as software maintenance and code review [7, 15]. Thus, prior work has focused on these two aspects, exploring existing patterns of confusion and possible coping strategies.

Confusion experienced during code review. Extensive studies have shown that high-quality code reviews are essential for maintaining code quality [23, 24] and fostering collaboration among team members [2, 29]. However, reviewers can experience confusion during the code review process, which hinders their ability to notice defects [10, 11]. Ebert *et al.* studied confusion in code reviews [9] by training a classifier using 396 general and 396 inline code review comments mined from the Android project and manually labeling the comments based on whether the reviewer expressed confusion. The classifier achieved substantial to high *precision* (up to 0.875 for general and 0.615 for inline comments). Their follow-up study [10] surveyed Android developers, identifying 30 reasons, 14 impacts, and 13 coping strategies for confusion. Moreover, they conducted an exploratory study [11] to investigate common causes of confusion, mapping 38 articles to reveal 13 solutions and 5 impacts for confusion during code review.

[1] https://github.com/openhab/openhab-addons
[2] https://github.com/confluentinc/ksql
[3] https://doi.org/10.5281/zenodo.11051281
[4] https://github.com/AlbertSGS/appendix-reevaluating

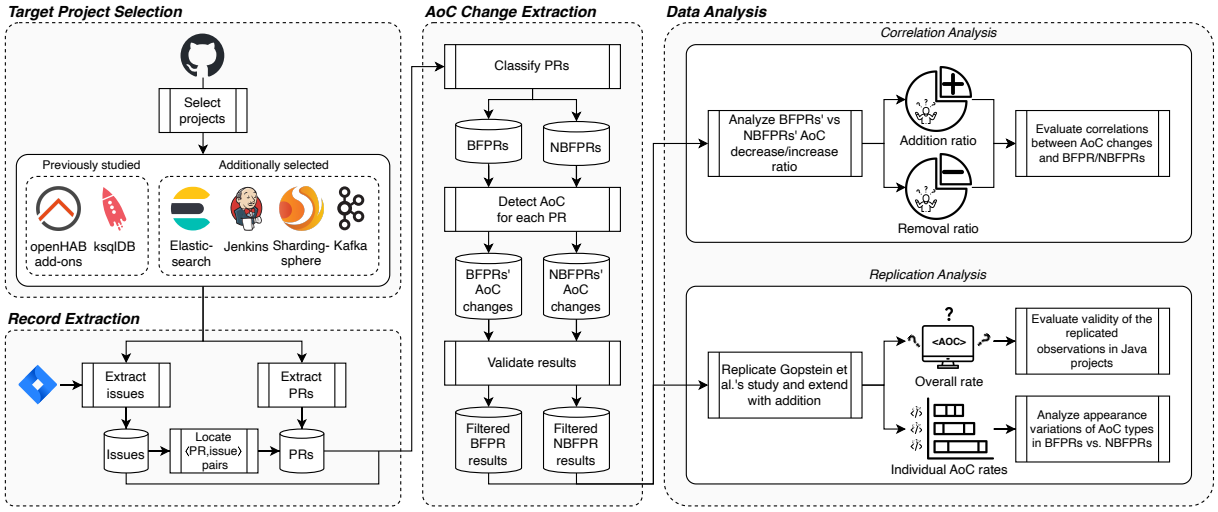

Figure 1: An overview of the study design.

Confusion experienced during development tasks. Compared to code review, identifying confusing code during development tasks is more challenging because (1) developers lack a clear process to denote confusing code, and (2) its long-term consequences, such as higher defect rates, are harder to detect. Prior work has explored possible sources for confusion and identified several factors, including code readability [12, 21], code smells [18, 30], and anti-patterns [1, 28]. While these studies did not investigate the concept of confusion directly, their common goal was to improve code comprehensibility and proactively prevent confusion. Only in the past decade have there been studies on confusion exclusively [7, 14]. De Mello *et al.* [7] studied confusing code's social representations among two developer groups, one closer to research, and the other closer to industry, revealing 11 common representations.

Atoms of Confusion. Gopstein *et al.* [14] identified 15 small code patterns in C-like languages named *Atoms of Confusion*, which could be a possible source of confusion for programmers. They found these patterns prevalent and defect-inducing in open-source C/C++ projects [15]. However, their follow-up study [13] showed that AoCs were not usually the cause of incorrect evaluations, and correct evaluations did not necessarily mean correct understanding.

de Oliveira *et al.* [8] reported the first association between physical signals and AoCs, employing an eye-tracking camera to monitor 30 students' eye movement. Their experiments showed that AoCs increased comprehension time by 43.02%, and that the subjects' gaze was more focused around AoCs. da Costa *et al.* [6] conducted similar experiments with 32 novices on six types of AoC in Python, seeing an increase in time spent and the number of answer attempts on code containing AoCs. Yeh *et al.* [33] further explored the impact of AoC on physical signals by analyzing the *Electroencephalogram* (EEG) measurements of individuals when reading code with and without AoCs. However, their results indicated that AoCs did not significantly increase the subjects' cognitive load in general.

Atoms of Confusion in Java. The concept of AoCs as small patterns introducing complexity to the code has since been studied extensively across different programming languages and contexts. Based on the set of 15 AoC types in C/C++ proposed by Gopstein *et al.* [14], Langhout and Aniche [22] presented 14 AoC types in Java. They surveyed 132 students, discovering that AoCs hindered the students' ability to comprehend the code, and that certain types of AoC confuse participants more than others. Mendes *et al.* [25] and Tahsin *et al.* [31] conducted empirical studies on the prevalence of AoCs in different sets of open-source Java projects. They found that specific types of AoC occurred frequently in their analyzed projects, while others were relatively rare. Furthermore, they found strong correlations between certain code- and project-related metrics, such as depth of inheritance tree, project age and types of maintenance task, and the prevalence of AoCs.

While there is mounting evidence of negative associations between AoCs and software maintainability measures in Java systems, recent work calls the impact of AoCs into question. For example, Pinheiro *et al.* [27] found out that there was no strong evidence to suggest that AoCs led directly to defect fixes. They showed that only 4.52% of their studied commits had removed AoCs. Their manual inspections of AoCs in defect-fixing and improvement commits showed that only 11.68% of the AoCs in those commits directly related to the fix or improvement. Moreover, Bogachenkova *et al.* [4] investigated the connection between the presence of AoCs and confusion in code review comments in open-source Java projects. The results showed a weak correlation between the presence of AoCs in code changes and the expression of confusion in inline comments. By comparing AoCs counts before and after PRs are closed, they observed that the majority of PRs did not remove AoCs.

While some of these findings align with prior work [15], they challenge the commonly perceived notion that AoCs are defect-inducing. These contradictions have motivated us to study this topic and test this assumption more thoroughly.

3 STUDY DESIGN

We set out to conduct an empirical study to investigate the relationship between AoCs and software quality. Figure 1 provides an overview of the design of our study, which comprises four

Table 1: Overview of the studied projects

	openHAB add-ons	kslqDB	Elasticsearch	Jenkins	Shardingsphere	Kafka
Application domain	Home automation	Database	Search engine	Automation server	Data sharding	Stream-processing
No. of studied PRs	7,960	5,566	42,891	7,033	9,742	6,451
No. of labels	41	80	588	44	75	22
No. of stars	1.7k	5.6k	64.8k	21.3k	19.2k	26.7k
No. of contributors	478	180	1,853	758	566	1,097
Java percentage	97.8%	99.7%	99.8%	85.6%	95.0%	78.5%

stages. We first perform (1) target project selection, where candidate projects are considered and appropriate projects are selected for further analysis. Next, for each selected project, we perform (2) record extraction, (3) AoC change extraction, and (4) data analysis. Below, we describe each stage of our study design.

3.1 Target Project Selection

We focus our evaluation on a selection of open-source projects that satisfy four inclusion criteria:

C1: Written primarily in Java — Java is one of the most frequently adopted programming languages on GitHub[5]. Prior studies on AoC have focused on systems that are written in Java [4, 22]. We choose to continue to focus on projects that are primarily implemented in Java to allow for a more direct comparison to the prior work. To this end, we require candidate projects to have the majority of their code written in Java. To operationalize C1, we select candidate projects that have at least 75% of their code written in Java.

C2: Maturity — GitHub is known to contain many projects that would be poor candidates for our study, such as personal projects not meant for wide use, student assignments, personal forks of existing projects, and abandoned projects. To mitigate the likelihood of such projects influencing our findings, we require candidate projects to be sufficiently popular from both observer and contributor perspectives. To operationalize C2, we select only projects that have at least 100 contributors and have more than 1,000 stars.

C3: Diversity of application domain — To improve the representativeness of our study, we decide to select a diverse set of software systems from different application domains. To ensure the quality of the dataset under study, we purposefully select candidate projects that are sampled from different domains. To operationalize C3, we browse through candidate projects selected with the previous criteria and select those that belong to different domains.

C4: High-quality labeling of PRs/issues — Since we need to identify which PRs are bug-fixing with high accuracy, we require that the subject systems accurately label their PRs/issues, i.e., the label of each PR/issue should reliably reflect whether it relates to a defect, an enhancement, or a new feature. To operationalize C4, we manually inspect the labels of a random sampling of 383 PRs/issues of each candidate subject system to verify that they are accurate.

Table 1 summarizes the details of the six selected projects. In addition to satisfying our inclusion criteria, we select the openHAB add-ons[1] and the ksqlDB[2] projects since they were also analyzed by Bogachenkova *et al.* [4], and thus provide a point of reference for comparison. We add the Elasticsearch,[6] Jenkins,[7] Shardingsphere,[8] and Kafka[9] projects to our set of studied projects to extend our observations to additional contexts.

3.2 Record Extraction

Pull Requests (PRs) are the primary means by which the studied projects evolve. Therefore, to address our research questions, we must extract and analyze a series of PRs and relevant issues. This stage is composed of three tasks:

Extract PRs — For each studied project, we select a list of PRs created before June 22, 2023. For each PR, we extract metadata, such as the PR number, associated labels, and links to related issue(s). The PR number serves as the input for our next stage, while labels and links are essential for PR classification.

Extract issues — Issue reports and their metadata, such as issue labels and links to PR(s), are extracted to identify bug-fixing PR(s).

Locate ⟨PR, issue⟩ pairs — For the openHAB add-ons and ksqlDB projects, we organize the extracted PR-issue links into ⟨PR, issue⟩ pairs. For the Jenkins and Kafka projects, we establish ⟨PR, issue⟩ pairs based on the JIRA issue keys that are explicitly referenced in the titles or descriptions of PRs.

3.3 AoC Change Extraction

To study the relationship between PR type and AoC changes, we need to classify PRs and extract their AoC changes. Figure 1 provides an overview of this stage, which comprises PR classification, AoCs detection, and validation steps.

Classify PRs — We classify the studied PRs into sets of *Bug-Fixing PRs* (BFPRs) and *Non-Bug-Fixing PRs* (NBFPRs).

In the openHAB add-ons, ksqlDB, Jenkins, and Kafka repositories, the PRs are explicitly labeled to indicate their type; however, contributors do not always label PRs accurately, potentially introducing noise if solely relied upon for classification. To mitigate this noise, we use the verified ⟨PR, issue⟩ pairs, and consider PRs that (1) are labeled "bug" and (2) are linked to issues labeled "bug" as BFPRs. PRs that do not meet both requirements are labeled as NBFPRs. Exceptions exist for the Jenkins and Kafka repositories, where PRs may not linked to Jira issues, but do contain issue keys (e.g. `JENKINS-10000`) in their metadata. In this case, we search for issue keys in the titles and descriptions of PRs and classify matched PRs as a BFPR if the referenced Jira issue has the "bug" label.

[5] https://github.blog/2023-11-08-the-state-of-open-source-and-ai/

[6] https://github.com/elastic/elasticsearch

[7] https://github.com/jenkinsci/jenkins

[8] https://github.com/apache/shardingsphere

[9] https://github.com/apache/kafka

The Elasticsearch and Shardingsphere projects require a different approach. Here, we classify PRs solely using PR labels because the projects encourage the use of accurate type-indicating labels. In the Elasticsearch repository, PRs labeled with >bug are classified as BFPRs, and the others are classified as NBFPRs. In the Shardingsphere repository, only PRs labeled `type:bug` are classified as BFPRs. PRs with other type-indicating labels are classified as NBFPRs, and the ones without any labels are ignored.

Detect AoCs for each PR — To study the changes in AoCs across PRs, we use a Java AoCs detection tool[10] that was developed based on Langhout and Aniche's study [22]. The tool extracts the types of AoC, the files in which the AoCs are located, and the locations within those files where AoCs appear. Additionally, we extend the detector to support our replication analysis with PR author metadata, the paths of modified files, and the number of added and removed lexer tokens per changed file. We also modify the tool's definition of *change of literal encoding*, *logic as control flow*, *omitted curly braces*, *repurposed variable*, and *type conversion* atoms to incorporate Pinheiro *et al.*'s definition [27] of Java AoCs. Finally, we omit the detection of *constant variable*, *arithmetic as logic* and *dead, unreachable, repeat* atoms, as they were found to be not confusing by both Langhout and Aniche [22] and Gopstein *et al.* [15]. The list of studied AoC types is available in our online appendix.[4] Our modified version of the tool is compiled and available online.[3] Of the 76,610 extracted PRs, 43,802 (57.2%) contain AoCs.

Validate results — To mitigate potential inaccuracies of the Java AoCs detector, we apply a filter to remove invalid tool output. We detail the identification of these records below.

Let PR_i be PR number i, and $\text{AoC}^i_{\text{before}}$, $\text{AoC}^i_{\text{after}}$, $\text{AoC}^i_{\text{added}}$, $\text{AoC}^i_{\text{removed}}$ and $\text{AoC}^i_{\text{untouched}}$ be total numbers of AoCs before PR_i, after PR_i, added, removed, and untouched by accepting PR_i, respectively. We validate the tool's output using Equation 1 below:

$$\begin{aligned} &\mathit{validate}\,(\text{PR}_i) \\ &:= (\text{AoC}^i_{\text{after}} - \text{AoC}^i_{\text{before}} = \text{AoC}^i_{\text{added}} - \text{AoC}^i_{\text{removed}}) \\ &\wedge (\text{AoC}^i_{\text{after}} - \text{AoC}^i_{\text{removed}} = \text{AoC}^i_{\text{untouched}}) \end{aligned} \tag{1}$$

We apply Equation 1 for each AoC type in each PR. If the expression evaluates to false for any AoC type, we exclude the PR from further analysis. This step results in discarding 2,328 out of 43,802 PRs (5.31%) to avoid inaccurate AoC values that could introduce noise in the analysis stage and hinder the reliability of this study.

3.4 Data Analysis

To study the correlation between AoC changes and PR type, and to revisit the findings of previous studies, we conduct two analyses:

Correlation analysis — The objective of the correlation analysis is to evaluate whether AoC changes and PR types share a quantitative association. We also revisit Pinheiro *et al.*'s study [27] in this section.

Table 2 shows the confusion matrix of the AoC changes per PR type. Variables a and b represent the numbers of BFPRs with a decreased and increased AoC cardinality after merging the PR, respectively, whereas c and d denote similar numbers for NBFPRs.

Replication analysis — To revisit the findings of Gopstein *et al.* [15] in the Java context, we begin with their replication package.

[10] https://github.com/SERG-Delft/atoms-of-confusion-detector

Table 2: The confusion matrix of AoC changes vs. PR types.

	AoC No. change	
PR type	decreased	increased
BFPR	a	b
NBFPR	c	d

The replication package produces the relative rates of AoC removal and addition for BFPRs and NBFPRs, as well as these relative rates for each individual AoC type, which include:

$$rrate_{rem} = \frac{rem_{\text{bfpr}}}{rem_{\text{nbfpr}}} \tag{2}$$

$$rrate_{add} = \frac{add_{\text{bfpr}}}{add_{\text{nbfpr}}} \tag{3}$$

where, e.g., $rem_{\text{bfpr}} := \frac{\text{AoC}^{\text{bfpr}}_{\text{removed}}}{N^{\text{bfpr}}_{\text{removed}}}$, and the other terms of the fractions in Equations 2 and 3 are defined similarly. In the example, $\text{AoC}^{\text{bfpr}}_{\text{removed}}$ is the number of AoCs removed by BFPRs, and $N^{\text{bfpr}}_{\text{removed}}$ is the number of tokens removed by BFPRs.

We divide the number of AoCs by the number of tokens to mitigate the impact of commit size. Gopstein *et al.* used "human-visible *Abstract Syntax Tree* (AST)" nodes as their choice of tokens while we use the lexer tokens as a replacement for the Java context, which provides a similar normalization of the size of the PR. We explain "human-visible AST" in our appendix in detail.[4]

4 STUDY RESULTS

Below, we describe the results of our empirical study with respect to our research questions. For each research question, we first describe the approach that we followed, then report the results that we observed, and finally we discuss the implications of our results.

RQ1: Are there any associations between the trend of AoC changes and PR type?

In this RQ, we study the relationship between the trend of AoC changes and PR type in the six selected Java systems. More specifically, we hypothesize that if merging BFPRs (NBFPRs) has a non-random association with decreasing the number of AoCs, it may imply that removing AoCs aids the process of fixing defects (enhancing code). Prior work has also explored similar hypotheses [15].

Approach. First, we set out to revisit the study of Pinheiro *et al.* [27] using our data set and an alternative statistical test. Specifically, they used the Mann-Whitney U test to study the relation between AoC removal rates and PR type, but we choose to use the χ^2 test to study the relation between the trend of AoC changes and PR type. We choose the χ^2 test since our dataset is limited to six projects with high-quality labeling. This constraint, stemming from our rigorous project selection process, renders Mann-Whitney U tests less powerful due to the small sample size [26]. A low-power statistical test would also increase the risk of Type I and Type II error [5].

To perform the χ^2 test, we organize the output of the AoC detection tool[10] into confusion matrices for the cases of the addition of AoCs and the PR type, as well as the removal of AoCs and the

Table 3: The confusion matrices of AoC changes vs. the PR type for the studied projects

	openHAB add-ons		ksqlDB		Elastic-search		Jenkins		Sharding-sphere		Kafka	
PR type	dec	inc	dec	inc	dec	inc	dec	inc	dec	inc	dec	inc
BFPR	105	1171	21	246	331	4768	41	828	34	374	73	1074
NBFPR	227	2363	172	1708	1688	17264	91	1309	645	4760	188	1993

Table 4: The ratios and χ^2 statistical test results for the studied projects. Bold indicates significance ($\alpha \approx 0.008$)

Metrics	openHAB add-ons	ksqlDB	Elastic-search	Jenkins	Sharding-sphere	Kafka
dec_{bfpr}	31.63%	10.88%	16.39%	31.06%	5.01%	27.97%
inc_{bfpr}	33.14%	12.59%	21.64%	38.75%	7.29%	35.02%
p-value	0.62	0.57	$\mathbf{3.97 \times 10^{-8}}$	0.09	0.04	0.03

Table 5: One-tailed Mann-Whitney U test statistics. Bold indicates significance ($\alpha = 0.008$)

Project	Statistic	p-value
openHAB add-ons	821,251.0	$\mathbf{6.11 \times 10^{-5}}$
ksqlDB	37,688.0	$\mathbf{3.25 \times 10^{-3}}$
Elasticsearch	13,599,801.5	$\mathbf{3.29 \times 10^{-35}}$
Jenkins	361,506.0	$\mathbf{1.58 \times 10^{-7}}$
Shardingsphere	81,443.5	$\mathbf{1.58 \times 10^{-2}}$
Kafka	667,316.5	$\mathbf{5.41 \times 10^{-7}}$

PR type. Holm-Bonferroni Correction is applied to the significance level, leading to $\alpha \approx 0.008$. If the test concludes that there is a relation between AoC changes and PR types, we compare the following ratios using the data in confusion matrices similar to Table 2 to determine the nature of this relation:

$$dec_{\text{bfpr}} \coloneqq \frac{a}{a+c}, \qquad inc_{\text{bfpr}} \coloneqq \frac{b}{b+d}. \tag{4}$$

dec_{bfpr} represents the ratio of BFPRs with AoC removal over all the PRs with AoC removal. Similarly, inc_{bfpr} is the ratio of BFPRs with AoC addition. If $dec_{\text{bfpr}} > inc_{\text{bfpr}}$, it implies that AoC removals are happening significantly more frequently in bug-fixing PRs. Similar to previous studies [15], we can then conclude that the AoCs contribute to the defect-proneness of the code.

Results. Table 3 and 4 provide the confusion matrices for the studied projects and the results of our statistical analyses, respectively. We observe that there is a larger percentage of BFPRs that increase the number of AoCs than those that decrease the number of AoCs for each project. Furthermore, for the Elasticsearch project, the χ^2 test indicates that there is a statistically significant correlation between the trend of AoC changes and PR type. However, the calculated ratios indicate that PRs with an increased number of AoCs are more frequent among bug-fixing PRs in these two projects, since the p-values of the χ^2 test are lower than the corrected α value. The results indicate that in the case of Elasticsearch, PR authors add AoCs more often than they remove them in BFPRs.

Discussion. Our findings suggest that the examined types of AoC are not associated with defect fixing. This conclusion is based on the expectation that if they were defect-inducing, they would be removed more frequently in BFPRs compared to NBFPRs—an observation that is not supported by our results, since the percentage of BFPRs that decrease the number of AoCs is lower than the percentage of BFPRs that increase them. The results of our χ^2 tests indicate that either there is no correlation between the trend of AoC changes, or in the case of the Elasticsearch project, they counterintuitively tend to *help* when fixing defects. This result contradicts the findings of prior work on C/C++ projects, which suggested that AoCs could be a primary reason for defects [15]; however, it complements the work of Pinheiro *et al.* [27] in Java projects, where they discovered that the rate of AoC removal in defect-fix commits was not greater than in other types of commits in 18 of the 21 projects that they studied. Thus, we conclude that AoCs in Java are not quantitatively associated with incidences of defects, as more BFPRs tend to increase the number of AoCs than NBFPRs.

To evaluate this further, we study the difference between the removal and addition of AoCs in each type of PR. To this end, we compare $\{add_{\text{bfpr}_1}, \ldots, add_{\text{bfpr}_n}\}$ with $\{rem_{\text{bfpr}_1}, \ldots, rem_{\text{bfpr}_n}\}$ for the list of BFPRs defined as $\{\text{bfpr}_1, \ldots, \text{bfpr}_n\}$, where add_{bfpr_i} and rem_{bfpr_i} are defined accordingly as shown in the example after Equations 2 and 3, respectively. We exclude the rates related to PRs that do not add or remove lines of code, as these actions result in a token count of zero for either addition or removal (i.e., $N_{\text{added}}^{\text{bfpr}_i} = 0$ or $N_{\text{removed}}^{\text{bfpr}_i} = 0$). Such cases, with zero denominators during the calculation of add_{bfpr_i} and rem_{bfpr_i}, are deemed irrelevant for our analysis. Since our goal is to test whether AoCs are added more often than they are removed in BFPRs, we use the one-tailed Mann-Whitney U test, where the null hypothesis (H_0) states that there is no significant difference between these two sets, and the alternative hypothesis (H_a) states that the first group (add_{bfpr_i}) tends to be larger than the second group (rem_{bfpr_i}).

Table 5 presents the one-tailed Mann-Whitney U test outcomes, where the Holm-Bonferroni correction sets $\alpha \approx 0.008$. We reject H_0 for all but the Shardingsphere project and conclude that the values of add_{bfpr_i} tend to be higher than the values of rem_{bfpr_i} for those

Table 6: Relative AoC removal and addition rates in BFPRs over NBFPRs

	Gopstein *et al.* [15]	openHAB add-ons	ksqlDB	Elasticsearch	Jenkins	Shardingsphere	Kafka
$rrate_{rem}$	1.21	1.18	1.32	1.47	1.46	1.82	1.19
$rrate_{add}$	1.28	1.21	1.24	1.11	1.14	1.36	1.11

Table 7: Two-tailed Mann-Whitney U test statistics for AoC addition rates. Bold indicates significance ($\alpha = 0.004$)

Project	Statistic	p-value
Gopstein *et al.*	635,162,714.5	**0.000**
openHAB add-ons	1,600,418.5	0.393
ksqlDB	240,950.5	0.415
Elasticsearch	47,487,787.5	0.171
Jenkins	596,487.0	0.155
Shardingsphere	962,619.0	0.186
Kafka	1,202,295.0	0.989

Table 8: Two-tailed Mann-Whitney U test statistics for AoC removal rates. Bold indicates significance ($\alpha = 0.004$)

Project	Statistic	p-value
Gopstein *et al.*	298,829,664.5	**0.000**
openHAB add-ons	1,464,817.0	0.711
ksqlDB	221,686.5	0.327
Elasticsearch	40,525,879.0	**0.001**
Jenkins	456,188.0	0.329
Shardingsphere	962,619.0	0.747
Kafka	1,082,670.5	0.327

projects. This provides further support to our conclusion that AoCs are not quantitatively associated with the incidences of defects.

While in all studied Java projects, the number of AoC in BFPR tends to increase more often than it decreases, our statistical analysis indicates a significant difference in only one of the studied projects. Thus, we conclude that AoC in Java are not inherently defect-inducing. This observation contradicts the findings of prior work on C/C++ projects, but complements prior work on Java projects, and suggests that further investigation is necessary to better understand the factors that are at play.

RQ2: Do PR types correlate with relative AoC removal and addition rates?

In this RQ, we study the association between the relative rate of AoC changes and PR type in Java.

Approach. We replicate Gopstein *et al.*'s [15] study with our dataset to determine whether one type of PR removes or adds AoCs more often. To this end, we organize the results from the Java AoC detection tool[10] into a replication-compatible format. Contrary to Gopstein *et al.* [15] and Pinheiro *et al.* [27], who solely focus on the removal of AoCs, our study expands the scope to include the addition of AoCs as well. By examining the relative rates of both AoC removal and addition, we strive to gain a more holistic perspective on their association, facilitating more informed conclusions.

To this end, the replication package begins by conducting a χ^2 test for each type of AoC change to examine the randomness of the relationship. This is followed by the calculation of the relative rates of AoC removal ($rrate_{rem}$) and addition ($rrate_{add}$), as defined by Equation 2 and 3. We infer that the type of AoC change plays a role in the defect-fixing process if one relative rate distinctly surpasses another, or if one value is greater than one and the other is less than one. If neither is the case, we cannot draw any conclusion.

Results. Table 6 provides the relative AoC removal and addition rates for each project. The χ^2 tests for all relative rates yield $p \ll 0.008$. In all of the studied projects, $rrate_{rem} > 1$ and $rrate_{add} > 1$, meaning that AoCs are both removed and added at a higher rate in BFPRs. The Shardingsphere and ksqlDB projects have the highest $rrate_{rem}$ and $rrate_{add}$ values. The openHAB add-ons, ksqlDB, Elasticsearch and Jenkins projects share similar relative rates.

Discussion. While our findings on the relative removal rates of the studied projects align with the previous study in C/C++ projects [15], we also observe that the relative addition rates are in the same direction as the relative removal rates, i.e., AoCs are added more often in BFPRs than in NBFPRs. Therefore, we cannot conclude that AoCs are defect-inducing unless we ignore the relative addition rates.

To test whether the ratio of AoC removal and addition rates are significantly associated with PR type, we conduct two sets of two-tailed Mann-Whitney U tests, one for the ratio of AoC addition rates and one for their removal rates, to see whether they differ significantly. We choose the two-tailed over the one-tailed variant in this case because our observations can be drawn irrespective of the direction of the association. In these experiments, we compare (1) $\{add_{\text{bfpr}_1}, \ldots, add_{\text{bfpr}_n}\}$ with $\{add_{\text{nbfpr}_1}, \ldots, add_{\text{nbfpr}_n}\}$, and (2) $\{rem_{\text{bfpr}_1}, \ldots, rem_{\text{bfpr}_n}\}$ with $\{rem_{\text{nbfpr}_1}, \ldots, rem_{\text{nbfpr}_n}\}$. Similar to RQ1, we exclude PRs that did not add or remove lines.

The results of the statistical tests are shown in Table 7 and 8. The statistical test results indicate that we cannot reject H_0 for all projects except for removal rates in the Elasticsearch project. Therefore, we cannot safely claim that the relative rates significantly differ between BFPRs and NBFPRs, i.e., removal and addition of AoCs are not to be quantitatively associated with PR type. While prior work on C/C++ projects was able to draw such a conclusion [14], our results suggest that these conclusions do not extend to systems that are primarily written in Java.

While the results of the χ^2 tests show that there is an association between AoC removal and PR type, considering AoC addition shows that we cannot attribute AoCs for defects as they have been both removed and added more often in BFPRs. Follow-up two-tailed Mann-Whitney U tests indicate that we cannot draw conclusions based on the relative rates in the Java projects.

Table 9: p-values for the χ^2 test results of $rrate_{add}$ and $rrate_{rem}$ for the 10 types of AoC in the studied projects

	openHAB add-ons		ksqlDB		Elasticsearch		Jenkins		Shardingsphere		Kafka	
Types	$rrate_{add}$	$rrate_{rem}$	$rrate_{add}$	$rrate_{rem}$	$rrate_{add}$	$rrate_{rem}$	$rrate_{add}$	$rrate_{rem}$	$rrate_{add}$	$rrate_{rem}$	$rrate_{add}$	$rrate_{rem}$
CLE	-1.75	-2.14	×	×	-1.63*	-1.99	-∞	-∞	5.72***	-∞	-∞	-∞
CO	1.22****	1.19***	1.14	1.05	1.14****	1.60****	-1.24*	1.17	1.13	1.60****	1.22****	1.34****
Ind	2.05	-∞	-∞	-∞	-1.78*	-1.11	2.10	-1.19	×	×	1.15	∞
IOP	1.21****	1.23****	1.51****	1.67***	1.15****	1.49****	1.35****	1.84****	1.60****	1.51***	1.08*	1.14*
LCF	1.37**	-1.10	-1.15	1.76*	-1.08	1.11	1.37*	-1.11	1.24	2.05**	-1.05	2.09****
OCB	×	3.90	×	×	-2.68	-∞	∞	2.94	×	×	1.57	2.83
PostID	1.37	-1.12	-1.67	-∞	1.01	1.31*	-1.12	-1.15	2.67**	2.74	1.03	-1.68
PreID	1.17	-1.03	-∞	-∞	-2.91***	-2.47*	1.37	1.96	4.81****	17.70****	1.54*	-1.89*
RV	1.03	-2.05	1.87	-∞	1.39	1.69*	2.75	3.91	-∞	-∞	-1.05	1.89
TC	-2.89**	-1.17	1.87	-∞	-1.39	-1.34	1.14	-∞	-∞	-∞	-2.79**	-4.96***

• The number of asterisk (*) next to the cell values indicates the power of 0.1 which the p-value is less than. E.g., n of * indicates $p < 0.1^n$.
• The × symbol indicates that there is no such AoC type added/removed in that project.

RQ3: Why are different types of AoC removed and added in bug-fixing and non-bug-fixing PRs?

In this RQ, we study each AoC type with respect to their relationship with PR type.

Approach. Following Gopstein *et al.*'s approach [15], we calculate whether the relation between AoC change and PR type differs for each AoC type. We categorize the AoCs based on the types proposed by Langout *et al.* [22] and then report whether any non-random association exists and what $rrate_{add}$ and $rrate_{rem}$ is for each type.

Results. Figure 2 visualizes $rrate_{add}$, $rrate_{rem}$, and the significance results of the χ^2 test for each of the studied projects per AoC type in the Elasticsearch project. The set of bar charts for all studied projects, with one for $rrate_{add}$ and $rrate_{rem}$ for each project, can be found in our online appendix.[3]

The relative rates when NBFPRs add or remove more than BFPRs are expected to be less than one. Such relative rates are inverted, and a negative notation is used for improved visual clarity. For example, for a relative rate of 0.80, we calculate $-1/0.80$ and write -1.25 instead. We adopt this notation in the complete set of relative rates that are presented in Table 9. Relative rate comparisons in this section are based on this notation.

We observe that for relative rates with $p < 0.01$—i.e., indicating statistical significance—some types of AoC exhibit specific patterns in terms of their relative removal and addition rates. Below, we summarize our observations.

O1 The *Change of Literal Encoding* (CLE) atom is neither added nor removed in the ksqlDB project and is only added and removed in NBFPRs in the Jenkins and Kafka projects.
O2 The *Conditional Operator* (CO) atom is added and removed significantly more in BFPRs in the openHAB add-ons, Elasticsearch and Kafka projects.
O3 The *Indentation* (Ind) atom is neither added nor removed in the Shardingsphere project, is only added and removed in NBFPRs in the ksqlDB project.
O4 The *Infix Operator Precedence* (IOP) atom is added and removed significantly more in BFPRs in all but the Kafka project.
O5 The *Omitted Curly Braces* (OCB) atom is neither added nor removed in the ksqlDB and Shardingsphere project.

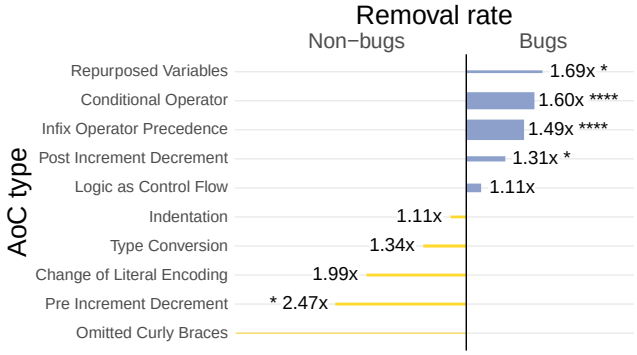

Figure 2: Relative removal rate in the Elasticsearch project.

O6 The *Pre Increment Decrement* (PreID) atom is added and removed significantly more in BFPRs in the Shardingsphere project.
O7 The *Repurposed Variable* (RV) atom is only added and removed in NBFPRs in the Shardingsphere project.
O8 The *Type Conversion* (TC) atom is added and removed significantly more in NBFPRs in the Kafka project.

For the remaining AoC types, either there is no discernible pattern, or the relative rates are not significantly different. We conjecture that developers add and remove these types unintentionally.

Discussion. We categorize our observations into three patterns.

P1 Multiple projects have significant relative rates. O2 and O4 belong to this group.
P2 One project has significant relative rates. O6 and O8 belong to this group.
P3 Exceptions, where either a specific AoC type is neither added nor removed, or both $rrate_{add}$ and $rrate_{rem}$ are $\pm\infty$, in some groups. O1, O3, O5, O7 belong to this group.

We discuss the observations in each group accordingly.

P1 — Pinheiro *et al.* [27] inspected 77 AoCs in defect-fixing and improvement commits to determine whether they relate to the commit. Of the 77 inspected AoCs, only 9 were found to have likely led to the commit. For each observation in P1, we follow a comparable approach. We sample ten BFPRs in each project that either remove

Table 10: Number of AoC changes for CLE, Ind, OCB, and RV atoms. BF = bug-fixing PRs, NBF = non-bug-fixing PRs.

	CLE				Ind				OCB				RV			
	Added		Removed		Added		Removed		Added		Removed		Added		Removed	
Repository	BF	NBF	BF	NBF	BF	NBF	BF	NBF	BF	NBF	BF	NBF	BF	NBF	BF	NBF
openHAB add-ons	5	36	3	25	1	2	-	12	-	-	1	1	1	4	1	8
ksqlDB	-	-	-	-	-	2	-	1	-	-	-	-	1	6	-	3
Elasticsearch	16	185	6	121	17	215	10	113	2	38	-	12	29	148	18	108
Jenkins	-	7	-	-	11	12	3	14	2	-	3	4	6	5	3	3
Shardingsphere	5	21	-	17	-	-	-	-	-	-	-	-	-	4	-	4
Kafka	-	35	-	12	2	5	1	-	2	9	3	4	3	9	2	4

more than add, or add more AoCs than they remove, and conduct a manual examination to determine whether the AoCs are intentionally removed or added for defect-fixing or improvement purposes, as appropriate. Similar to Pinheiro *et al.*, our decisions are based on PR contexts, including the title, description, and comments of a PR, as well as the related issue report and code changes.

In O2, we observe that $rrate_{add}$ and $rrate_{rem}$ for the *Conditional Operator* (CO) atom are greater than 1. We then search for ten BFPRs from the openHAB add-ons, Elasticsearch, and Kafka projects where this AoC type is either removed or added. We inspect 30 BFPRs, containing 62 CO AoCs, of which 37 contain null checks in their conditions. In the two AoCs in the Elasticsearch project, test files are the primary source of the change, and the conditions contain `randomBoolean()` for random testing. In either case, the removal and addition of these AoCs are not due to their confusing nature, but rather their utility for defect-fixing and testing. The rest of the CO AoCs are neither added nor removed because they are confusing.

Regarding observation O4, which indicates that the *Infix Operator Precedence* (IOP) atom is added and removed significantly more in all but the Kafka projects, we sample ten BFPRs from each project that either removed or added this AoC type. In total, we examine 50 PRs, collectively containing 95 IOP AoCs. Unlike the example instance presented by Langhout and Aniche [22], 82 of the examined AoCs are of the format: `if (a != null || a.equals("str"))`. This format is recognized as an IOP atom due to two consecutive operators (`!=` and `||`) without clarifying parentheses for operator precedence. For some IOP AoCs, null checks are removed as they are no longer necessary, or added to avoid `NullPointerExceptions`. In other cases, `instanceof` checks are explicitly added. The instances of IOP AoCs under scrutiny are unlikely to be the sole reason for these changes. The remaining 12 of the IOP AoCs include mixed mathematical operations without parentheses, e.g., `a + b * c`, and they are not found to be the direct cause of their associated PRs.

Overall, the observations in P1 indicate that the *Conditional Operator* and *Infix Operator Precedence* AoC are not considered primary causes of defects. Rather, many AoCs are modified due to their usefulness in defect-fixing and testing, rather than their potential to introduce defects. This finding supports the conclusion that AoCs may not be inherently increasing defect proneness, or that developers do not perceive them as confusing.

P2 — The observations in this pattern concern a single project and a single AoC type. Hence, we study PRs from that project containing changes to AoCs of that AoC type.

O6 highlights that both $rrate_{rem}$ and $rrate_{add}$ are greater than 1 for the PreID atom in the Shardingsphere project. To investigate, we examine BFPRs from the project that either removes more AoCs of PreID than it adds, or adds more AoCs than it removes. Three PRs meet this criterion, totaling five PreID AoCs added and 20 AoCs removed. Interestingly, the PreID AoCs are either `++sequenceID` or `++currentSequenceID`. The first two PRs made small, unrelated changes near these AoCs, while the last PR underwent a large refactoring and removed all occurrences of `sequenceID` and `currentSequenceID` thereby eliminating all related PreID AoCs.

O8 reveals that $rrate_{rem}$ and $rrate_{add}$ are both less than -1 for the TC atom in the Kafka project. Hence, we search for NBFPRs from the project that either remove more TC AoCs than they add or add more than they remove. 43 PRs meet this criterion, with 53 TC AoCs being added and 62 being removed. Of the 115 TC AoCs, 97 are typecasting from `int` to either `short`, `byte` or `char`. It appears that they could be avoided, as PR #11393[11] removed these occurrences by changing the input data type from `int` to `short`, with the PR author claiming:

> "We're removing more casts than we're adding."

However, in PR #12135,[12] a developer justified setting the input data type to `int`, stating:

> "This is because Java's default type is int – i.e. line[s]... above would take the value 0/1 as int. So we basically need to either do the conversion in each line[s]... above, or just do the conversion once here."

These conversations show that the removal and addition of TC occurrences in these cases are not due to their confusing nature. Typecasting could be necessary, and the developers are attempting to minimize its usage. Another interesting case where TC is removed is to avoid Java 20's lossy conversions warning.[13]

In summary, project- and language-specific factors including project style guides and implicit type conversion may influence changes to certain AoC types. Our online appendix contains more examples to elaborate on this point in detail.[4] Further investigation is encouraged to understand how these and other similar factors affect the prevalence and perceived confusion of AoCs.

P3 — These AoC types are either untouched or only affected in one type of PR in some projects, while in others, the relative rates do

[11] https://github.com/apache/kafka/pull/11393
[12] https://github.com/apache/kafka/pull/12135
[13] https://github.com/apache/kafka/pull/13582

not show significance. Intuitively, the *Indentation* (Ind) and *Omitted Curly Braces* (OCB) atoms, for example, are rarely seen or changed, as Java IDE software can identify and remove them. Similarly, the *Change of Literal Encoding* (CLE) and *Repurposed Variables* (RV) atoms occur slightly more frequently but are still rare compared to more prevalent types because developers would likely avoid them in the first place. To verify this claim, we count the number of the observed AoC types that are removed and added in BFPRs and NBFPRs respectively, presented in Table 10. We can see that these AoC types are not prevalent across all the projects we examined.

Our inspection reveals that certain AoC types are modified not because they are confusing, but rather due to their frequent use in defect-fixing and testing. Project- and language-specific factors can also contribute to these modifications. Additionally, several AoC types are actively avoided and are therefore not prevalent.

5 THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of our study, categorizing them according to the framework proposed by Wright *et al.* [32]. We describe threats to the construct, internal, and external validities, and the steps we take to mitigate them.

Construct Validity. Construct validity assesses the extent to which a test or tool measures the intended construct. In our study, the Java AoC detection tool[10] developed by the Software Engineering Research Group at TU Delft[14] is one of the potential sources of construct validity threats. While the tool has been developed by a renowned research group and employed in prior research [4], we modify the tool's definition of several AoC types to incorporate prior studies [25, 27], and we implement an additional validity check (Equation 1) to improve the tool's accuracy.

Another threat to construct validity is the categorization of PRs using their linked issues, as classifying PRs into bug-fixing and non-bug-fixing is a non-trivial task. It is known that BFPRs are not usually linked with defects [3]. To mitigate this threat, we randomly sample PRs from all studied PRs in the projects and label enough data points to have a 95% confidence interval.

Finally, Gopstein *et al.* [14] stated that AoCs were not defects themselves. It is possible that even if an AoC leads to the injection of a defect, it is not removed after the defect is fixed, and vice versa. Our method fails to capture this scenario, but it is beyond our scope.

Internal Validity. Internal validity concerns whether confounding factors exist in our study. Since confusion is a qualitative and subjective attribute and is not entirely measurable, it is impossible to determine all the factors contributing to whether AoCs are confusing to developers. We mitigate this threat by following the footsteps of previous works [15, 27], investigating individual AoC types, and uncovering the reasons behind their changes. Certainly, there are various other qualitative factors related to our observation, but they are beyond the scope of this study.

External Validity. External validity concerns the generalizability of our results. Two of our six studied projects were previously examined by Bogachenkova *et al.* [4]. Furthermore, we ensure that the other four projects we select for our study have high-quality labeling. These projects also offer diversity in size and application, enriching our study's scope. Our choice for the number of studied projects is constrained by two main factors: (1) the time-intensive process of extracting AoC changes for each PR using the GitHub REST API,[15] and (2) the reliability of PR/issue labels used to classify the PRs. Although we deliberately select projects to ensure diversity in category and size, we acknowledge that these may not comprehensively represent all existing projects. Nevertheless, the selected projects indicate a broad spectrum of application types and scales. Thus, the implications of our study should be considered with an understanding of its scope by both researchers and practitioners.

6 CONCLUSION

Our study aims to determine whether *Atoms of Confusion* are defect-inducing in Java software development. We mine 76,610 PRs from six open-source Java projects and conduct statistical analysis to investigate whether a non-random relationship exists between the trend of AoC changes (increase and decrease) and the PR type (bug-fixing and non-bug-fixing). Additionally, we replicate Gopstein *et al.*'s study [15] and extend it by incorporating the addition of AoCs—which has not been previously considered—to explore whether the removal and addition of AoCs are connected to different PR types. There are three key findings from our study:

- **Bug-fixing PRs do not decrease statistically more than increase the number of AoCs.** No conclusive evidence indicates that AoCs are defect-inducing. Incorporating AoC addition to Gopstein *et al.*'s study shows that we cannot blame AoCs for defects because they are added, as they are removed, more often in BFPRs than NBFPRs.
- **Modification to certain AoC types could be due to their usefulness in defect fixing or code improvement.** In reality, these AoCs are frequently used, making them less confusing. Examples can be found in our online appendix.[4]
- **Certain project- and language-related factors could contribute to AoC change.** Senior developers tend to follow their or the projects' coding conventions, even if they will introduce AoCs.[4] It is possible that these conventions are initially confusing to junior developers, and this calls for further investigation.

Our research challenges the assumption that AoCs inherently confuse developers in Java projects, highlighting the necessity for more in-depth investigations into the specific conditions under which AoCs become perplexing. Based on our findings, we propose to assess the factors that affect the confusing nature of AoCs, by exploring other projects and programming languages. We also plan to address the shortcomings of our work by conducting a complementary survey study. When investigating subjective concepts such as confusion, incorporating people's views into the data analysis will provide valuable insights to our study. Future research should not only look into the prevalence and effect of, but also practitioners' opinions on and coping strategies for, the *Atoms of Confusion*.

ACKNOWLEDGMENTS

This work is partially funded by the Waterloo-Huawei Joint Innovation Lab.

[14] https://github.com/SERG-Delft

[15] https://docs.github.com/en/rest

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, New York, NY, USA, 181–190. https://doi.org/10.1109/CSMR.2011.24

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, USA, 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[3] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 121–130. https://doi.org/10.1145/1595696.1595716

[4] Victoria Bogachenkova, Linh Nguyen, Felipe Ebert, Alexander Serebrenik, and Fernando Castor. 2022. Evaluating Atoms of Confusion in the Context of Code Reviews. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, NY, USA, 404–408. https://doi.org/10.1109/ICSME55016.2022.00048

[5] Robert M Christley. 2010. Power and error: increased risk of false positive results in underpowered studies. *The Open Epidemiology Journal* 3, 1 (2010), 16–19. https://doi.org/10.2174/1874297101003010016

[6] José Aldo Silva da Costa, Rohit Gheyi, Fernando Castor, Pablo Roberto Fernandes de Oliveira, Márcio Ribeiro, and Baldoino Fonseca. 2023. Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. *Empirical Software Engineering* 28, 4 (18 May 2023), 81. https://doi.org/10.1007/s10664-023-10311-0

[7] Rafael de Mello, José Aldo da Costa, Benedito de Oliveira, Márcio Ribeiro, Baldoino Fonseca, Rohit Gheyi, Alessandro Garcia, and Willy Tiengo. 2021. Decoding Confusing Code: Social Representations among Developers. In *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, New York, NY, USA, 11–20. https://doi.org/10.1109/CHASE52884.2021.00010

[8] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael de Mello, Anderson Oliveira, Alessandro Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. 2020. Atoms of Confusion: The Eyes Do Not Lie. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/3422392.3422437

[9] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2017. Confusion Detection in Code Reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, NY, USA, 549–553. https://doi.org/10.1109/ICSME.2017.40

[10] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2019. Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, New York, NY, USA, 49–60. https://doi.org/10.1109/SANER.2019.8668024

[11] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2021. An exploratory study on confusion in code reviews. *Empirical Software Engineering* 26 (2021), 1–48. https://doi.org/10.1007/s10664-020-09909-5

[12] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 286–296. https://doi.org/10.1145/3196321.3196347

[13] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 605–616. https://doi.org/10.1145/3368089.3409714

[14] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 129–139. https://doi.org/10.1145/3106237.3106264

[15] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 281–291. https://doi.org/10.1145/3196398.3196432

[16] J.D. Herbsleb and A. Mockus. 2003. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering* 29, 6 (2003), 481–494. https://doi.org/10.1109/TSE.2003.1205177

[17] J.D. Herbsleb and D. Moitra. 2001. Global software development. *IEEE Software* 18, 2 (2001), 16–20. https://doi.org/10.1109/52.914732

[18] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/ICPC.2016.7503706

[19] Johannes Hofmeister, Janet Siegmund, and Daniel V. Holt. 2017. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, New York, NY, USA, 217–227. https://doi.org/10.1109/SANER.2017.7884623

[20] Ivar Jacobson, Grady Booch, and James Rumbaugh. 1999. *The Unified Software Development Process*. Addison-Wesley Professional. https://isbnsearch.org/isbn/9780201571691

[21] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An Empirical Study Assessing Source Code Readability in Comprehension. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, NY, USA, 513–523. https://doi.org/10.1109/ICSME.2019.00085

[22] Chris Langhout and Maurício Aniche. 2021. Atoms of Confusion in Java. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, New York, NY, USA, 25–35. https://doi.org/10.1109/ICPC52881.2021.00012

[23] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 192–201. https://doi.org/10.1145/2597073.2597076

[24] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

[25] Wendell Mendes, Oton Pinheiro, Emanuele Santos, Lincoln Rocha, and Windson Viana. 2022. Dazed and Confused: Studying the Prevalence of Atoms of Confusion in Long-Lived Java Libraries. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, NY, USA, 106–116. https://doi.org/10.1109/ICSME55016.2022.00018

[26] Gottfried E Noether. 1987. Sample size determination for some common nonparametric tests. *J. Amer. Statist. Assoc.* 82, 398 (1987), 645–647. https://doi.org/10.1080/01621459.1987.10478478

[27] Oton Pinheiro, Lincoln Rocha, and Windson Viana. 2023. How They Relate and Leave: Understanding Atoms of Confusion in Open-Source Java Projects. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 119–130. https://doi.org/10.1109/SCAM59687.2023.00022

[28] Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. 2020. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology* 122 (2020), 106278. https://doi.org/10.1016/j.infsof.2020.106278

[29] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/3183519.3183525

[30] Vallary Singh, Lori L. Pollock, Will Snipes, and Nicholas A. Kraft. 2016. A case study of program comprehension effort and technical debt estimations. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, New York, NY, USA, 1–9. https://doi.org/10.1109/ICPC.2016.7503710

[31] Noshin Tahsin, Nafis Fuad, and Abdus Satter. 2023. Prevalence of 'Atoms of Confusion' in Open Source Java Systems: An Empirical Study. (Feb. 2023). https://doi.org/10.22541/au.167570695.54470176/v1

[32] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. 2010. Validity Concerns in Software Engineering Research. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (Santa Fe, New Mexico, USA) *(FoSER '10)*. Association for Computing Machinery, New York, NY, USA, 411–414. https://doi.org/10.1145/1882362.1882446

[33] Martin K-C Yeh, Yu Yan, Yanyan Zhuang, and Lois Anne DeLong. 2022. Identifying program confusion using electroencephalogram measurements. *Behaviour & Information Technology* 41, 12 (2022), 2528–2545. https://doi.org/10.1080/0144929X.2021.1933182