How Does Experience Influence Developer Perceptions of Atoms of Confusion?

Guoshuai (Albert) Shi
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
albert.shi@uwaterloo.ca

Farshad Kazemi
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
f2kazemi@uwaterloo.ca

Shane McIntosh
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
shane.mcintosh@uwaterloo.ca

Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
migod@uwaterloo.ca

***Abstract*—Background: *Atoms of Confusion* (AoCs) are small, syntactically valid code patterns that can increase cognitive load during program comprehension. Although prior studies have suggested that AoCs are prevalent and potentially harmful, recent work has scrutinized their practical impact, especially in real-world development settings.**

Objective: This confirmatory study aims to investigate whether developer experience influences how AoCs are perceived and repaired during code comprehension. We investigate whether developers of varying experience levels behave differently by examining both comprehension time and the repair they prefer when interacting with code that contains AoCs.

Method: We propose a two-phase study consisting of a pre-screening questionnaire and a ~~controlled~~ between-subject experiment. Participants will complete ten comprehension tasks, each featuring a defective snippet of code containing a different AoC, followed by a choice between three functionally equivalent repairs that vary in AoC inclusion. We will analyze task time using linear regression and repair preference using multinomial logistic regression, with developer experience as the key *Independent Variable* (IV).

***Index Terms*—code confusion, ~~controlled~~ between-subject experiment, empirical software engineering**

I. Introduction

Code that is difficult to understand slows ~~down~~ software maintenance, increases the likelihood of defects, and reduces team productivity [1, 2]. While many readability concerns stem from structural or semantic complexity, confusion can also arise from syntactic patterns that are valid yet ~~difficult~~ hard for humans to parse. In particular, Gopstein *et al.* identified a set of such patterns, which they called *Atoms of Confusion* (AoCs) [3].

Prior work has identified and catalogued AoCs in programming languages such as C, C++, and Java [4, 5]. Empirical studies have investigated the prevalence of these patterns in open-source codebases [4, 6], the degree to which they slow down comprehension [3, 7, 8], and their defect proneness [4]. Prior studies have yielded weak or inconsistent effects when settings vary [9, 10, 11]. This raises the question: When and for whom are AoCs problematic?

In this study, we investigate whether developer experience predicts behavioural responses to AoCs. We hypothesize that experienced developers will tolerate syntactic shorthand that some may find terse, whereas novices may favour clear, understandable, but potentially longer code. To study this hypothesis, we combine a pre-screening questionnaire with a ~~controlled~~ between-subject experiment involving ten tasks, each targeting a specific AoC. Participants will select one of three repairs: one containing the AoC that the task targets, one without that AoC, and one with additional AoCs that make the repair shorter.

We analyze outcomes using regression modeling with the goal of disentangling whether AoCs are universally confusing or selectively problematic depending on experience. Should our study confirm our hypothesis, the findings will inform code review practices, refactoring tools, and readability guidelines that better account for developer experience levels.

II. Background and Related Work

The concept of *Atoms of Confusion* (AoCs)—syntactic patterns that may introduce confusion—was introduced by Gopstein *et al.* [3], who identified 15 such patterns in C-like languages. Follow-up work found these patterns to be both prevalent and defect-prone in open-source C/C++ systems [4]. Listing 1 shows an example from the FreeBSD repository, where the developer confused the order of precedence between the bitwise OR and ternary operators, which introduced a defect. More recent studies have shown that the presence of AoCs in code snippets was not necessarily why the participants incorrectly evaluated the outcome, nor did correct evaluations guarantee understanding of AoCs [12].

```
ulpmc->cmd = htobe32(V_ULPTX_CMD(ULP_TX_MEM_WRITE) |
     is_t4(sc) ? F_ULP_MEMIO_ORDER :
    F_T5_ULP_MEMIO_IMM);
```

Listing 1. Defective code snippet with Infix Order Precedence AoC

Subsequent research investigated the behavioral and physiological effects of AoCs. De Oliveira *et al.* [7] used eye tracking to show that AoCs slowed comprehension time by over 40% and drew focused visual attention. Da Costa *et al.* [8] reported that novices took more time to read code and needed more attempts when fixing Python code that contained AoCs.

Researchers have also studied AoCs that can occur in Java programs. Langhout and Aniche [5] proposed 14 types of AoCs for Java, and surveyed 132 students, showing that some patterns hinder comprehension more than others. Mendes *et al.* [6] and Tahsin *et al.* [13] analyzed open-source Java projects and found that while some AoCs types were common, others were rare. Their presence correlated with project-level factors, such as the depth of the inheritance tree, the project age, and indicators of maintenance activity.

Prior work has also scrutinized the impact of AoCs. Yeh *et al.* [14] found no significant increase in cognitive load (measured using EEG signals) between code with and without AoCs. Pinheiro *et al.* [10] found that only 4.5% of commits explicitly removed AoCs, and just 11.7% of those removals were the direct cause of fixes or improvements. Bogachenkova *et al.* [9] observed only weak correlations between reviewer confusion and the presence of AoCs, with most *Pull Request*s (PRs) retaining them after review. Shi *et al.* [11] found no consistent link between AoCs and defect-fixing activity in the PRs of six Java projects. In fact, some bug-fixing PRs introduced more AoCs than they removed.

While early findings supported the idea that AoCs negatively affect code quality [4], recent work challenges this assumption. This contrast highlights a gap in our understanding and motivates deeper empirical investigation into how developers interpret and respond to AoCs. We hypothesize that these contradictory findings of past work may in part be explained by the expertise levels of developers. Experts may have a higher tolerance for AoCs than novices. The confirmatory study that we propose will evaluate this hypothesis.

III. Research Questions and Hypotheses

This study investigates how developer experience influences the perception and debugging of code containing AoCs. We focus on the time developers take to read and repair defective code with AoCs, compared to a baseline task without one, and their choice between three semantically equivalent repairs that differ in terms of the presence or absence of AoCs.

We operationalize developer experience as a continuous variable (i.e., years of programming in ~~the participant's primary language~~Java), collected in the pre-screening questionnaire. While treated as continuous, this variable will likely be discretized ~~in practice~~ due to self-reported, rounded inputs. Each task is treated independently, as it represents ~~different types~~ a different type of AoCs. We investigate two research questions:

RQ1 Does developer experience affect the time taken to read and repair defective code that contains an AoC?
$H1_A$: More experienced developers will show a smaller increase in task time when completing tasks with AoCs compared to the baseline task.
$H1_\emptyset$: Developer experience is not associated with the task time when completing tasks with AoCs compared to the baseline task.

RQ2 Does developer experience influence the type of repair preferred for defective code with an AoC?
$H2_A$: The type of repairs that developers prefer is associated with their experience levels.
$H2_\emptyset$: Developer experience does not correlate with the types of repairs developers prefer for AoC-prone code.

These hypotheses are grounded in the assumption that experienced developers are more tolerant of syntactic complexity and are more willing to trade clarity for conciseness. Conversely, less experienced developers may prefer repairs that are easier to read. However, it is also possible that novice developers might prefer the repair that minimally alters the defective code, even if it retains an AoC, while experienced developers, drawing on prior exposure to real-world codebases, may favour clearer alternatives that improve long-term maintainability. Given these competing intuitions, we do not commit to a specific direction in our second hypothesis.

IV. Study Design

This study investigates how developer experience influences the perception and debugging of code containing *Atoms of Confusion* (AoCs). To capture ~~both~~ background variability and behavioral data, the study ~~is divided into~~ has two phases: a **pre-screening questionnaire** and a **~~controlled~~ between-subject experiment**. ~~We intend to release all anonymized data to facilitate~~ They will be conducted online via custom web interface. All anonymized data will be released via Zenodo to support replication.

A. Pre-screening Questionnaire

The pre-screening questionnaire ~~is used to collect~~ collects participant demographics and programming background ~~and~~ to ensure that only qualified participants continue to the ~~controlled~~ between-subject experiment. There are three categories of questions (Table I):

- **Qualifying questions**, which establish eligibility and are used to exclude participants who do not meet the inclusion criteria;
- ***Independent Variable* (IV) question**, used to retrieve the key IV for analysis; and
- **Follow-up questions**, used to support post-hoc interpretation or exploratory analysis.

Qualifying Criteria. Participants must meet specific criteria to reduce variation caused by confounding factors and improve internal validity. These criteria were chosen based on prior empirical studies of code comprehension [5, 7].

To qualify for the experiment, participants must:

TABLE I
PRE-SCREENING QUESTIONNAIRE ITEMS AND THEIR OPERATIONALIZATION.

Question	Type	Operationalization
Category A. Qualifying Questions		
Q1. What is your primary spoken language?	Binary	English or Non-English.
Q2. What ~~is your highest level of education? Ordinal High school diploma, Bachelor's degree, Master's degree, PhD, or other. Q3. What~~ was your field of study?	Nominal	Software engineering, Electrical engineering, Mathematics, or other.
~~Q4~~Q3. Have you received a formal education in programming?	Binary	Yes or No.
~~Q5~~Q4. How many years have you been programming professionally?	Ordinal (numeric)	A number between 0 and 100, inclusive.
~~Q6~~Q5. Which of the following best describes your current role in software development?	Nominal	Software developer, QA engineer, DevOps engineer, researcher, or other.
~~Q7~~Q6. How many hours per week do you work on OSS projects on average?	Ordinal	A number between 0 and 168, inclusive.
~~Q8~~Q7. What is your ~~primary~~ most proficient programming language?	Nominal	Java, Kotlin, Python, C, C++, Rust, ..., or other.
Category B. *Independent Variable* (IV) Question		
~~Q9~~Q8. How many years have you been programming in ~~language chosen in Q8~~Java?	Ordinal (numeric)	A non-negative integer no larger than [answer of ~~Q5~~Q4].
Category C. Follow-up Questions		
~~Q10. Which programming languagesdo you use regularly~~Q9. What are your more frequently used programming languages?	Multiple choice	Java, Kotlin, Python, C, C++, Rust, ..., and others.
Q.10. How many years have you been programming in [each programming language selected in Q9]?	Ordinal (numeric)	A non-negative integer no larger than [answer in Q4].
Q11. How often do you perform code reviews?	Ordinal	Daily, Weekly, Monthly, Rarely, Never.
Q12. Do you follow coding style guides in your projects?	Binary	Yes or No.
~~Q14~~Q13. How do you usually approach understanding unfamiliar code?	Nominal	Refer to documentation, Test/debug the code, Discuss with colleagues, or other.
~~Q15~~Q14. How much time do you typically spend reading code versus writing it?	Ordinal	Mostly reading, Balanced, or Mostly writing.

- speak or have proficiency in English (to avoid misinterpretation due to language processing difficulties);
- ~~hold at least a Bachelor's degree (to ensure a baseline level of academic training);~~
- have studied software engineering, computer science, or a related field;
- have received a formal education in programming; and
- have at least one year of programming experience.

These conditions are intended to filter out participants who might be confused by syntax, task framing, or language semantics unrelated to the presence of AoCs.

Independent variable question. The key IV in our study is the participants' self-reported years of experience programming in ~~their primary language~~Java. We operationalize this as a continuous variable, which allows us to investigate experience as a gradient rather than a binary (novice vs. expert) construct.

Follow-up Questions. Although not used in the primary analysis, we collect additional background information such as review frequency and approach to unfamiliar code. These may inform future exploratory analyses or help explain variability in comprehension strategies.

B. ~~*Controlled*~~ *Between-Subject* *Experiment*

In the second phase, participants complete ten tasks, each associated with one type of AoCs. For each task, we present participants with a defective Java code snippet and ask them to select ~~one~~ their most preferred repair from three alternatives. We present an example task in our online appendix[1]. Each task should take approximately 5-10 minutes for a developer with at least one year of Java experience. The tasks should be domain-neutral and center on general-purpose programming constructs (e.g., loops, conditionals, expressions) to isolate syntactic effects rather than domain knowledge.

We record the order in which each task is presented to participants. This allows us to model potential fatigue or learning effects that may arise over the course of the session. Task order will be included as a covariate in our statistical

[1]https://github.com/AlbertSGS/AoCs-experience_appendix

models to control for systematic variation due to position, and to explore whether comprehension behavior shifts across time.

Each repair was constructed to satisfy four conditions:

- all three versions repair the defect correctly;
- one repair removes the original AoC, often by rewriting the expression using more verbose logic;
- one repair retains the same AoC, correcting only the functional error; and
- one repair introduces different AoCs, yielding a shorter version compared to the previous two.

This design reflects real-world refactoring tradeoffs, where some developers may favour brevity, while others may prioritize clarity and readability.

Before the ten main tasks, we will display one task with a defective code snippet and three repairs *without* any AoCs. This task serves as a baseline to account for general differences in debugging ability across participants. We assume that experienced developers repair defects more quickly in general, and without this baseline, such effects could confound our interpretation of the impact of AoCs. ~~By comparing performance~~ The baseline task shares the same difficulty level with the tasks with AoCs. By comparing completion time on AoC-related tasks relative to the baseline, we can more accurately isolate the added difficulty introduced by AoCs.

The tasks are presented using a *custom-built web interface*. This interface randomizes the task and repair preference order to reduce order effects and logs two key behavioral outcomes:

- The time taken to interpret and respond to each task (measured from snippet display to repair selection); and
- The repair selected by the participant.

No time limits are enforced; participants may proceed at their own pace. However, they are encouraged to take breaks between tasks since we will record the time to resolve each task. Each task is self-contained and includes only the minimal code required to observe the effect of the AoC in question.

After the tasks, we will display ten follow-up questions asking them to justify each choice they made. We believe that a clarifying question will provide valuable insights to whether developers prefer clarity or brevity.

This within-participant design improves efficiency by collecting multiple observations per participant, with each task treated as an independent data point. This is appropriate because each AoC poses a distinct syntactic pattern and comprehension challenge. We therefore model tasks individually rather than aggregating across them.

~~*Pilot Study*~~

C. *Pilot Study*

Before full deployment, we will conduct a pilot study ~~for our study design~~ using convenience sampling (e.g., labmates or colleagues within our research group)~~. We plan~~ , aiming to recruit at least ten participants~~for the pilot. If pilot results reveal unclear task instructions or extreme task difficulty, we will revise or replace the affected items. This will~~ . The pilot will help assess the clarity of task instructions, web interface functionality, logging accuracy, ~~and task timing.~~ task timing, and the comprehensibility of code snippets. We will carefully check for unintentional cues in the code—such as naming, formatting, or spacing—that could bias participant choices, especially given the syntactic subtlety of AoCs. If pilot results reveal unclear instructions, overly difficult tasks, or biased code presentation, we will revise or replace the affected items. Insights from the pilot will ~~inform minor usability adjustments~~guide minor usability improvements, which will be documented transparently. No pilot data will be included in the final analysis.

V. Participants

The target population comprises software developers ~~using~~ identifying Java as their ~~primary~~ most proficient programming language. Participants will be recruited through professional networks, mailing lists, academic contacts, and developer communities. To be eligible, participants must have at least one year of programming experience and ~~use~~ identify Java as their ~~primary~~ most proficient programming language. They must also complete the pre-screening questionnaire. Participants who fail embedded attention checks, submit incomplete responses, or indicate no exposure to Java will be excluded from the analysis.

Participants are expected to complete all 10 tasks—each based on a distinct type of AoCs—in one session, although no time limit is imposed. We will offer incentives for those that complete all tasks. For each task, we collect two *Dependent Variables* (DVs): (1) the time taken to solve the task, and (2) the selected repair from the three available options.

Our IV is developer experience, measured as a continuous variable in years. We aim to analyze whether developer experience is associated with these outcomes.

Sample Size

We determined our target sample size based on power analyses for both DVs: task completion time (DV1) and repair preference (DV2).

DV1: Task Completion Time. We will analyze the relationship between developer experience and time to solve each task using separate task-level regressions. Although each participant completes all ten tasks, we treat them as distinct comprehension scenarios and do not model them as repeated measures. Each AoC presents a unique syntactic challenge, and comprehension of one does not directly inform another.

We estimated the required sample size using a standard F-test power analysis for linear regression with one continuous IV. Assuming a medium effect size ($f^2 = 0.0625$), $\alpha = 0.05$, and 80% power, the theoretical minimum is 3 participants per regression. However, this ~~reflects~~ assumes ideal conditions—linearity, normality, and homoscedasticity—which are unlikely~~to hold in our data~~.

We acknowledge that the relationship between developer experience and task time may be non-linear or non-monotonic.

Rather than assuming normality, we prepare for flexible model classes, such as:

- robust linear regression with heteroskedasticity-consistent standard errors,
- generalized additive models,
- quantile regression, or
- nonparametric correlation analysis (e.g., Kendall's τ).

To ensure these flexible models have sufficient support and are not overly sensitive to noise or skew, we conservatively estimate that at least **50–60 participants** are required for task-level regressions under realistic conditions.

DV2: Repair Preference. Repair preference will be modeled using multinomial logistic regression [15], with developer experience as a continuous IV. Each task presents three mutually exclusive repair options. As no closed-form power calculator exists for this configuration, we use an approximation strategy:

- We first estimated the lower bound using a chi-square test with three categories and medium effect size ($w = 0.3$), yielding **107 participants**.
- We then adjusted this to reflect the increased complexity of using a continuous IV in multinomial logistic regression. Following recommendations from Hsieh *et al.* [16], we applied a conservative inflation factor of 1.75, yielding a recommended total of **187 participants**.

Alternative Considerations and Final Decision.

We considered expanding our model to include additional IVs (e.g., code review frequency or comprehension style). However, modeling multiple IVs or their interactions (e.g., 3 experience levels × 3 behavioral categories) would substantially increase the required sample size. Preliminary estimates suggest such designs would require at least 248 participants, which is infeasible given current resource constraints.

We also considered treating developer experience as a categorical variable with three levels (~~e.g.,~~ novice, intermediate, expert)~~. This approach would allow clearer comparisons across defined experience groups and relax assumptions of linearity~~ , allowing clearer group comparisons and relaxing linearity assumptions. However, a between-subjects design with ~~3~~ three groups requires larger~~and more balanced samplesto detect group-level differences reliably~~, balanced samples. For DV1, ~~using an ANOVA-style~~ ANOVA power analysis [17] with medium effect size (~~$f = 0.25$), $\alpha = 0.05$~~$f = 0.25$), $\alpha = 0.05$, and 80% power ~~, the required total sample size is approximately~~ yields a required sample of about **159 participants**. This ~~does not include the additional~~ excludes the added complexity of repair preference modeling, which would ~~further increase this requirement if applied within each~~ increase the sample size further if applied per group.

Given these trade-offs, we retain a continuous formulation of developer experience as our primary IV. This ~~approach~~ preserves power under realistic sample sizes and supports flexible modeling (e.g., splines or robust regression) to capture potential non-linear effects without ~~inflating~~ increasing the sample size burden.

We therefore evaluated three realistic sampling scenarios:

- **Baseline power target**: 107 participants (based on chi-square approximation)
- **Fully powered MLR scenario**: 187 participants (to support multinomial regression with a continuous IV)
- **Pragmatic compromise**: 110–120 participants (sufficient for task-level regression and marginal MLR modeling)

We target a final sample size of **110 participants**, which balances empirical rigor with recruitment and budget feasibility. This sample size exceeds the requirement for linear modeling under relaxed assumptions and approaches adequacy for repair preference modeling with one continuous IV.

If resources allow, we will continue data collection up to 187 participants. If the final sample falls short, we will document the limitation and apply sensitivity analyses or bootstrapped confidence intervals to assess the robustness of our findings.

VI. Analysis Plan

This section ~~describes~~ outlines our statistical analysis ~~procedures for evaluating~~ for the two dependent variables: task completion time and repair preference. ~~To~~ We test our hypotheses ~~, we model~~ by modeling each task independently ~~using regression techniques suited for each outcomevariable~~with regression methods appropriate to each outcome.

A. Task Completion Time Delta (DV1)

For each of the ten tasks, we will model the relationship between task completion time and developer experience using **ordinary least squares (OLS) linear regression**.

- **Model.** Let Δt_{ij} be the delta time for participant i on task j, and x_i is their programming experience. We test $\boldsymbol{H1_A}$ by examining whether β_1 is significantly negative across tasks. Equation 1 models the test described.

$$\Delta t_{ij} = \beta_0 + \beta_1 x_i + \epsilon_{ij} \tag{1}$$

  β_0 is the expected value of y when $x = 0$.
- **Assumptions.** Linearity, homoscedasticity, and normal residuals will be checked.
- **Adjustments.** If residuals exhibit skew, heteroskedasticity, or non-linearity, we will consider:
  - *Robust regression* with HC3 standard errors,
  - *Generalized additive models (GAMs)* for nonlinear fits,
  - *Quantile regression* to examine median behavior,
  - or *Kendall's* τ as a nonparametric alternative.
- **Multiple testing.** We will analyze each task independently and correct for multiple comparisons using Holm–Bonferroni or false discovery rate (FDR).

Each regression will test $\boldsymbol{H1_A}$: whether experience predicts faster comprehension (shorter task time).

B. Repair Choice (DV2)

We will use **multinomial logistic regression** to analyze whether developers' experience predicts which repair they select for each task.

- **Model:** $\mathbb{P}(y_i = k) = \frac{\exp(\beta_{0k}+\beta_{1k}x_i)}{\sum_{j=1}^{3}\exp(\beta_{0j}+\beta_{1j}x_i)}$ for $k \in \{1,2,3\}$
- **Reference class:** We will use the "clean no-AoC" repair as the baseline class.
- **Predictor:** Experience (continuous)
- **Inference:** We will report odds ratios, 95% confidence intervals, and p-values.
- **Model checking:** We will inspect class balance and goodness-of-fit using pseudo-R^2, residual diagnostics, and likelihood-ratio tests.

If convergence or separation issues arise, we will apply penalized maximum likelihood estimation (e.g., ridge regression or Firth correction).

We record task order as a covariate to account for fatigue or learning effects that may influence task time. For exploratory purposes, we may pool responses across tasks to model aggregate trends in repair preference, although our primary analysis remains task-level.

C. Exploratory Analyses

We plan to explore whether self-reported code-reading habits, code review frequency, or preferred comprehension strategy correlate with task time or repair preference. These analyses will be clearly labeled as exploratory and not used to draw confirmatory conclusions.

- ~~Q10~~**Q9.** *Which programming languages do you use regularly?* **and Q10.** *How many years have you been programming in [each programming language selected in Q9]?* This may offer insight into how diversity in programming language shapes comfort with different coding styles or syntactic conventions, particularly with AoCs.
- **Q11.** *How often do you perform code reviews?* Responses may help inform post hoc exploration of whether review frequency relates to attention to stylistic clarity.
- **Q12.** *Do you follow coding style guides in your projects?* This may help contextualize whether coding style aligns with certain repair preferences, such as selecting clearer or more formally structured code.
- ~~Q14~~**Q13.** *How do you usually approach understanding unfamiliar code?* We may explore whether preferred comprehension strategies correspond to task behavior, such as time spent or interaction with different repair types.
- ~~Q15~~**Q14.** *How much time do you typically spend reading code versus writing it?* Participants' responses may clarify how reading-heavy workflows relate to sensitivity to syntactic features like AoCs.

We will also report descriptive statistics for each task (e.g., average time, repair selection proportions) to contextualize differences across AoCs.

Any groupings derived post hoc (e.g., binning participants into experience tertiles) will be labeled as exploratory. These analyses will not support confirmatory claims and are intended solely for hypothesis generation or interpretation.

VII. Expected Outcomes and Interpretation

A. Task Completion Time Delta (RQ1)

We expect that injecting AoCs into defective code will increase comprehension time for most developers. However, we hypothesize that this increase—the delta relative to ~~a baseline task—will~~ baseline—will be smaller for more experienced developers. If linear models reveal a consistent negative association between experience and delta time across tasks, this would support $\boldsymbol{H1_A}$. ~~If no such pattern emerges, or if effects are inconsistent, it~~ Inconsistent or null effects would support the null hypothesis.

B. Repair Preference (RQ2)

We expect repair preference to vary with experience:

- More experienced developers may prefer concise repairs, even if they include more AoCs.
- Less experienced developers may favour clearer, longer repairs that eliminate confusing patterns.

If multinomial regression shows that the probability of selecting a repair ~~containing~~ with more AoC increases with experience (relative to ~~selecting~~ the no-AoC repair), this ~~would support~~ supports $\boldsymbol{H2_A}$.

If no significant trend is observed, or ~~if~~ the no-AoC repair is equally preferred across experience levels, it ~~would suggest~~ suggests that developer preferences are not influenced by experience ~~in the way we~~ as hypothesized.

Alternative Interpretations and Null Results

If neither DV is significantly associated with experience, several explanations are possible:

- Developers may not find AoCs confusing enough to affect comprehension or repair preference.
- The differences between repair options may not be salient enough to drive a consistent behavioral effect.
- Other factors—such as ~~individual~~ reasoning strategies or code context—may play a stronger role than experience alone.

We will interpret null results cautiously and use exploratory follow-up analyses to generate hypotheses for future work.

Our goal is not to generalize across all possible confusing patterns but to better understand the nuanced role developer experience plays in resolving confusion in code.

VIII. Threats to Validity

~~We address potential threats to the validity of our study.~~

A. Internal Validity

Fatigue ~~or inattentiveness~~and learning effects: Participants complete ten tasks in a single sitting, which may ~~lead to cognitive fatigue~~ cause fatigue or learning effects as they become familiar with the interface or patterns. We mitigate ~~this~~ these by randomizing task order and allowing participants to ~~take the experiment~~ proceed at their own pace. ~~We will examine response time distributions to identify and~~ Task order will be recorded and used as a covariate, and we will examine response times to exclude implausibly fast submissions.

~~Learning effects:~~ ~~Although tasks are independent, familiarity with the interface or pattern types may improve performance over time. Task order randomization helps counterbalance this effect across participants.~~

Instrumentation bias: All participants use the same web-based interface, which controls code presentation and logs interaction times uniformly. Timing is captured using client-side JavaScript events to ensure consistency.

B. Construct Validity

Developer experience measurement: We use years of programming in ~~the participant's primary language~~ Java as a proxy for experience. While this measure is common in empirical software engineering, it may not capture the quality or depth of experience. We mitigate this by applying eligibility filters and considering exploratory covariates such as code review frequency and reading habits.

Task realism: Tasks are short and synthetic by design to isolate specific AoCs. While this improves control and internal validity, it may limit realism. However, similar task designs have been used in prior controlled studies on AoCs [3, 5, 7].

Repeated measures bias: Our within-subject design exposes each participant to multiple tasks, which may introduce intra-individual dependencies. While each task targets a distinct AoC and is modeled independently, this approach may not fully capture participant-level variance. As a robustness check, we will compare results using mixed-effects models with random intercepts for participants.

C. External Validity

Generalizability to broader developer populations: Our inclusion criteria may bias the sample toward relatively educated, Java-proficient developers. We make no claims about how developers from different backgrounds, languages, or experience levels would respond. Future replications can expand eligibility to compare trends across developer subgroups.

Ecological validity: In real development scenarios, comprehension occurs in richer contexts (IDE, version control, documentation). This experiment captures isolated comprehension episodes, which is appropriate for identifying fine-grained effects of syntactic patterns.

D. Conclusion Validity

We apply appropriate statistical models for each dependent variable and justify our sample size through formal power analysis and realistic constraints. Model assumptions will be verified, and alternatives used where necessary (e.g., robust regression, nonparametric methods). All confirmatory analyses will be clearly distinguished from exploratory ones.

IX. Conclusion

This registered report outlines a two-part confirmatory study ~~to examine~~ on how developer experience affects the comprehension and debugging of code ~~containing~~ with *Atoms of Confusion* (AoCs). We combine a pre-screening questionnaire with a ~~controlled~~ between-subject task-based experiment to evaluate two dependent variables: task completion time and repair preference.

Our hypotheses ~~focus on~~ test whether more experienced developers are faster and more tolerant of AoCs, while less experienced ~~developers~~ ones may favour readability over brevity. We ~~plan to~~ use task-level regression and multinomial modeling to ~~test~~ evaluate these claims.

By focusing on syntactically isolated ~~but~~ yet semantically realistic code examples, this study ~~contributes~~ offers new insight into ~~the behavioral effects of~~ how AoCs ~~and~~ interact with developer experience. The results may inform future readability metrics, ~~code~~ quality tools, and training materials that account for ~~developer background~~ experience when flagging or refactoring potentially confusing code.

References

[1] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "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*, MSR 2014, (New York, NY, USA), p. 192–201, Association for Computing Machinery, 2014.

[2] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, (New York, NY, USA), p. 286–296, Association for Computing Machinery, 2018.

[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*, ESEC/FSE 2017, (New York, NY, USA), p. 129–139, Association for Computing Machinery, 2017.

[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*, MSR '18, (New York, NY, USA), p. 281–291, Association for Computing Machinery, 2018.

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

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

[7] B. de Oliveira, M. Ribeiro, J. A. S. da Costa, R. Gheyi, G. Amaral, R. de Mello, A. Oliveira, A. Garcia, R. Bonifácio, and B. Fonseca, "Atoms of confusion: The eyes do not lie," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, (New York, NY, USA), p. 243–252, Association for Computing Machinery, 2020.

[8] J. A. S. da Costa, R. Gheyi, F. Castor, P. R. F. de Oliveira, M. Ribeiro, and B. Fonseca, "Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension," *Empirical Software Engineering*, vol. 28, p. 81, May 2023.

[9] V. Bogachenkova, L. Nguyen, F. Ebert, A. Serebrenik, and F. Castor, "Evaluating atoms of confusion in the context of code reviews," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (New York, NY, USA), pp. 404–408, IEEE, 2022.

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

[11] G. Shi, F. Kazemi, M. W. Godfrey, and S. McIntosh, "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*, pp. 154–164, 2024.

[12] D. Gopstein, A.-L. Fayard, S. Apel, and J. Cappos, "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*, ESEC/FSE 2020, (New York, NY, USA), p. 605–616, Association for Computing Machinery, 2020.

[13] N. Tahsin, N. Fuad, and A. Satter, "Prevalence of 'atoms of confusion' in open source java systems: An empirical study," Feb. 2023.

[14] M. K.-C. Yeh, Y. Yan, Y. Zhuang, and L. A. DeLong, "Identifying program confusion using electroencephalogram measurements," *Behaviour & Information Technology*, vol. 41, no. 12, pp. 2528–2545, 2022.

[15] C. Kwak and A. Clayton-Matthews, "Multinomial logistic regression," *Nursing research*, vol. 51, no. 6, pp. 404–410, 2002.

[16] F. Y. Hsieh, D. A. Bloch, and M. D. Larsen, "A simple method of sample size calculation for linear and logistic regression," *Statistics in medicine*, vol. 17, no. 14, pp. 1623–1634, 1998.

[17] T. K. Kim, "Understanding one-way anova using conceptual figures," *Korean journal of anesthesiology*, vol. 70, no. 1, p. 22, 2017.