Would a Rose by any Other Name Smell as Sweet? Examining the Cost of Similarity in Identifier Naming

Naser Al Madi
Computer Science
Colby College
nsalmadi@colby.edu

Matianyu Zang
Computer Science
Brown University
matianyu_zang@brown.edu

Abstract

Background: Identifier naming is one of the main sources of information in program comprehension, where the majority of software development time is spent. When reading natural language texts or code, readers perform lexical access to retrieve the orthography (word shape), phonology (pronunciation), and semantic (meaning) representations of words from memory. The successful retrieval of these representations is vital for success in comprehension and subsequent code maintenance and evolution.

Objective: This paper examines the cost of identifier similarity in orthography, phonology, or semantic representation and how that affects debugging performance and programmer workload. By recognizing common identifier naming combinations that hinder code comprehension, we can discover new programming best-practices and create automated tools that flag problematic naming combinations.

Method: Through a human experiment (n=43), we explore the impact of orthographic, phonological, and semantic similarity on debugging success, time, and workload. In our experiment, participants worked on debugging three programs, each of which has two versions that are identical except for one pair of identifiers, with either similar identifier names (e.g. i and j) or dissimilar names (e.g. row and column). Participants were randomly assigned a version of the code, and their performance was recorded to measure debugging success and time. At the end of each trial, they reported the subjective workload through NASA Task Load Index (NASA-TLX).

Results: We found some differences in debugging success and duration between similar and dissimilar identifiers with advanced programmers, but the differences are not statistically significant.

Conclusion: The results call for further investigation of identifier similarity and its influence on code comprehension. The study of identifier similarity can shed light on new linguistic anti-patterns that could potentially hinder code comprehension.

1. Introduction

Software developers spend the majority of their time reading source code for program comprehension, debugging, modifying, or learning (Maalej et al., 2014; Shneiderman and Mayer, 1979; Von Mayrhauser and Vans, 1995). In fact, several studies revealed that developers spend more than 50% of their time searching for information (Ko et al., 2007; Murphy et al., 2006; Maalej et al., 2014; Von Mayrhauser et al., 1997; Standish, 1984; Tiarks, 2011). Therefore, program comprehension is a critical component of the software development process, and even a small improvement in comprehension time can lead to significant gains in software development time and programmer workload. In addition, the association between identifier naming and code quality has been well established on the class and method levels (Butler et al., 2010). This is especially important when we consider the economic standpoint that software maintenance is the biggest factor in the cost of a software system (Barry et al., 1981; Siegmund and Schumann, 2015).

In programming, comprehension depends on a number of factors such as code comments and the names programmers choose for variables, classes, functions, modules, parameters, and constants, also known as identifier naming. In fact, identifier naming is regarded as one of the primary sources of information that programmers use to understand source code (Lawrie et al., 2006; Newman et al., 2020; Maletic and Marcus, 2001). Multiple studies have been conducted on identifier naming, and we group these previous works under two categories that are relevant to our current work: First, works that focused

on the influence of identifier naming on comprehension. Second, works that focused on the impact of identifier naming on workload and difficulty.

Under the first category, an early study by Takang et al. (1996) focused on the effects of comments and identifier names on code comprehension, and found that "programs that contain 'full' identifier names are more understandable than those with abbreviated identifier names." The study used multiple-choice questions and subjective code assessment to measure program comprehension, and one of the study's recommendations was to use a better assessment method of program comprehension in controlled experiments. Similarly, Lawrie et al. (2007) compared the influence of single letters, abbreviations, and full word identifiers on comprehension. The study found evidence that full word identifiers lead to better code comprehension than single letters and abbreviations. Another study on identifier abbreviations by Scanniello et al. (2017) compared full-word identifiers to abbreviations in a bug fixing task, and found no significant difference in debugging duration or effectiveness between the two. Moreover, Binkley et al. (2013) reported multiple experiments comparing camel case and underscore identifier naming styles. The study demonstrated that beginner programmers benefited from the use of camel case in comprehension and effort, while experienced programmers were not affected by identifier styles.

Under the second category, a key study by Arnaoudova et al. (2013) presented linguistic anti-patterns, which described common poor naming and commenting choices in code. Linguistic anti-patterns highlight inconsistencies between identifier names and their behaviors. A recent study by Fakhoury et al. (2020) focused on the influence of linguistic anti-patterns on developers' cognitive load during bug localization tasks. The study revealed that linguistic anti-patterns significantly increased developers' workload.

In this paper, we focus on the influence of identifier similarity on code comprehension and programmer workload. Identifier similarity could occur in one of three lexical dimensions, orthography (word shape), phonology (word pronunciation), and semantics (word meaning). When programmers use identifiers that are similar in one of the three dimensions, it is possible for confusing identifier pairs to hinder code comprehension. An example of orthographically similar identifiers that are often used in the same scope are 'i' and 'j' in nested loops. The two identifiers are similar in shape, easily causing programmers to substitute one for another or use them interchangeably, and that leads to logical errors that are difficult to localize and fix.

A previous study Aman et al. (2021) indicated that such confusing identifier naming combinations do occur in production software, yet an empirical study focusing on the effect of similarity in identifier naming on comprehension is still needed. The potential influence of identifier similarity on comprehension could uncover new linguistic anti-patterns that impede code comprehension.

The foundations of this study come from Cognitive Science, where the role of lexical access in reading natural language texts has been studied extensively (Rayner, 1998). Lexical access describes the retrieval of word shape, pronunciation, and meaning from memory during reading for comprehension. Typically, readers retrieve meaning and sound from word form, but occasionally top-down processing takes place in predictable contexts, for example, where the word is recognized from its meaning before the eyes make a fixation on it (Rayner, 1998). The presence of this top-down phenomena is usually associated with skilled readers who are more likely to skip predictable and frequent words (Rayner et al., 2006).

Despite sharing many cognitive processes with reading natural language texts, reading source code differs fundamentally in purpose, syntax, semantics, and viewing strategy from natural texts (Busjahn et al., 2015; Liblit et al., 2006; Busjahn et al., 2014; Schulte et al., 2010). This motivates the study of identifier naming and lexical similarity in code further, especially with the important implications on code comprehension and workload discussed earlier.

In this study, to avoid the methodological limitations of multiple choice questions, we focus instead on debugging (fixing logical errors in code) as an indicator of code comprehension. This approach allows us to measure debugging success probability and debugging duration as objective ways to evaluate code

comprehension under different conditions. In addition, we use NASA Task Load Index (NASA-TLX) to assess subjective workload during the debugging process. We aim to answer the following research questions:

- RQ1: Does identifier similarity influence success in debugging source code?
- RQ2: Does identifier similarity influence source code debugging time?
- RQ3: Does identifier similarity influence source code debugging difficulty?

Studying identifier similarity in source code helps uncover new best practices that enhance programmer productivity and comprehension. Such best practices can be integrated in automated tools for enhancing code readability, maintainability, and quality.

2. Pilot Study

In this section, we present the details of our first **IRB approved** pilot study and the sources of data we used to answer our research questions. Our within-subject pilot study consisted of 12 participants in total.

2.1. Objective

Our goal in the pilot study was to empirically verify the hypothesis that identifier similarity influences debugging success and time. We examined this hypothesis with orthographic, phonological, and semantic similarity in identifier naming, with the assumption that similarity could potentially hinder lexical access and in turn comprehension. We state two research questions to clarify our objective in the pilot study:

- RQ1: Does identifier similarity influence success in debugging source code?
- RQ2: Does identifier similarity influence source code debugging time?

By providing some initial results for the two questions in the pilot study, we aimed to gain insights for a larger experiment with additional sources of data and additional research questions (experiment one).

2.2. Participants

The 12 participants in our study were Computer Science students with experience ranging from one to three years. All participants had completed their second Computer Science course (CSII or equivalent) and were thus familiar with Java programming language. All participants were above the age of 18, and 5 were Female and were 7 Male. Participation was voluntary, and participants were awarded a $15 gift card after the experiment.

```
for(int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++){
                if (this.myGrid[j][j] == ""){
                        empty++;
                }
        }
}
```

(a) Similar

```
for(int row = 0; row < 3; row++){
        for (int column = 0; column < 3; column++){
                if (this.myGrid[column][column] == ""){
                        empty++;
                }
        }
}
```

(b) Dissimilar

Figure 1 – Pilot study orthographic similarity material (identifiers highlighted).

2.3. Material

As stimuli for our pilot study, we prepared three programs in Java that included a pair of identifiers similar in orthography, phonology, or semantics. The three programs had one logical defect (bug) each

that inhibited the code from working as intended when running. Nonetheless, programs could be compiled successfully without any syntactic errors. As a control, we modified the programs to use identifiers that were dissimilar, with all other aspects of the programs remaining identical to the first three programs. Therefore, each of the three programs had two versions, one with similar identifiers and one with dissimilar identifiers.

For orthographic similarity, we prepared a text-based Tic-Tac-Toe program in Java. The program consisted of 200 lines of code, and it included a logical defect in the check for a tie method. Figure 1 shows the two versions of similar and dissimilar identifiers presented. The similar version uses the identifiers 'i' and 'j' which are orthographically similar (in form), and the dissimilar version of the same code uses 'row' and 'column' instead. The rest of the code is identical. Both programs included the same logical error, seen at line three in Figure 1, where the grid was indexed incorrectly. Orthographic similarity was determined by programmatically comparing the shape of letters or words to determine the percentage of overlap.

```
public static String join(String separator,
                          String[] input) {
...
public static String connect(String str1, String str2){
```

(a) Similar

```
public static String listToString(String separator,
                                  String[] input) {
...
public static String AddSpace(String str1, String str2){
```

(b) Dissimilar

Figure 2 – Pilot study semantic similarity material (identifiers highlighted).

For semantic similarity, we prepared a phonebook program that stored data in a comma-separated file. One functionality of the program was allowing adding data from a new comma-separated file to the phonebook. The program was 149-line-of-code long. Figure 2 shows the similar and dissimilar versions of the program. The identifiers in this task were method names, and the logical error was using the incorrect method later in the code. The identifiers 'join' and 'connect' were semantically similar, while 'listToString' and 'addSpace' were dissimilar, and they performed the same job in the code. Semantic similarity was determined using Wordnet (Miller, 1995).

```
for(int numIter=0; numIter<100; numIter++) {
        monitor.roll();

        if ((numIter / 10) == 0) {
                monitorTens.add(monitor.getSideUp());
        }
}
```

(a) Similar

```
for(int iter=0; iter<100; iter++) {
        die.roll();

        if ((iter / 10) == 0) {
                monitorTens.add(die.getSideUp());
        }
}
```

(b) Dissimilar

Figure 3 – Pilot study phonological similarity material (identifiers highlighted).

For phonological similarity, we prepared a text-based Die simulation where a die was thrown one hundred-times, and every tenth throw was added to a list of results. Figure 3 shows similar and dissimilar identifiers in the same context in the program, where 'numIter' and 'monitor' are phonologically similar and 'iter' and 'die' are not. Both programs included the same logical error in the way results were added to the results list. Phonological similarity was determined using Wordnet (Miller, 1995).

2.4. Pilot Study Procedure

This experiment was conducted remotely using Zoom video conferencing and screen sharing. One member of the research team met with one participant on Zoom at a time, and the general procedure of the experiment was explained along with an online consent form. If the participant agreed to proceed

with the experiment, the online form collected the participant's programming experience in years along with demographic information.

The participant shared their screen with the member of the research team, and the participant was provided the experiment material. Each experiment included three programs in a random order to eliminate the order effects, and the programs were randomized in terms of similar and dissimilar identifiers as well. This means that each participant was given either two programs with similar identifiers and one that was dissimilar, or two programs with dissimilar identifiers and one similar. This accounted for the experience effect, where participants perform better in a repeated task, and also guaranteed that each participant worked on debugging programs with similar and dissimilar identifiers.

Each of the three debugging tasks had a maximum time of 15 minutes, and comments at the top of each program explained the intended function of the program, an example of the intended output, and the time limit of each task, but no information of the number of bugs was provided. Participants worked on writing and running the code in a natural software development environment using the VS Code Integrated Development Environment, and their behavior was recorded in terms of debugging time and whether they were successful in debugging or not. The experiment took approximately 50 minute in total, and after each task the code was saved and uploaded back to the research team.

2.5. Pilot Study Design

To answer our questions on the influence of identifier similarity on debugging success and time, we compared the debugging time and success probability of both similar and dissimilar versions of the same program. Since the only variable was identifier similarity, this could give us evidence in the form of increased or reduced time and success probability in the similar condition. In addition, we could focus on a specific type of similarity, such as orthographic, phonological, or semantic and inspect debugging success probability and time in each type separately.

2.6. Pilot Study Results

RQ1: Does identifier similarity influence success in debugging source code?

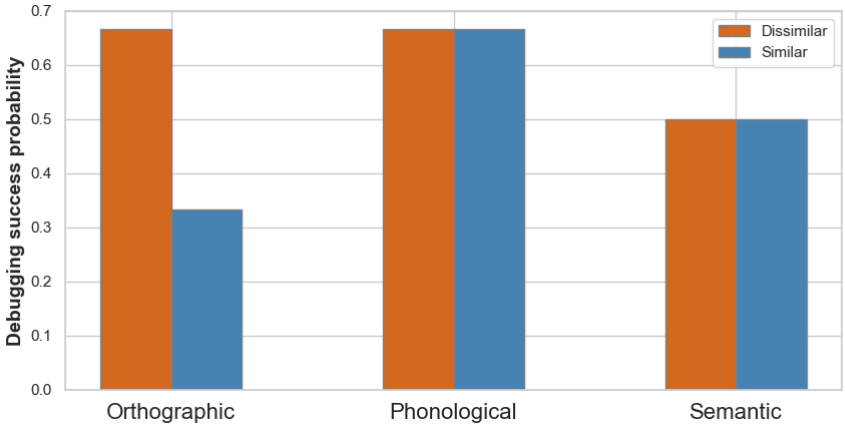

Figure 4 – Debugging success probability in the pilot study.

We measured debugging success probability as the proportion of participants who correctly fixed the logical error in a trial within 15 minutes. Figure 4 compares the success probability of similar and dissimilar identifiers. The figure shows that identifier similarity in phonology and semantics do not appear to influence debugging success, as similar and dissimilar identifiers result in the same debugging success probability. On the other hand, presenting orthographically similar identifiers to programmers decreased their aggregate debugging success from 66% to 33%. Yet, a Fisher's test of independence was performed to examine the relation between identifier orthographic similarity and debugging success. The relation between these variables was not significant, $p = .56$. Fisher's test was chosen since the sample size is small (i.e., $n < 50$).

RQ2: Does identifier similarity influence source code debugging time?

We measured the amount of time needed to successfully debug code with similar and dissimilar identifiers, in order to quantify the influence of identifier similarity on debugging time. Figure 5 shows debugging time (in seconds) of code with similar and dissimilar identifiers. The white dot is the median, the upper and lower limits are the maximum and minimum values respectively, and the shape is a histogram of the data points. The results demonstrate less debugging time for code with orthographically and phonologically dissimilar identifiers compared to the same code with similar identifiers. There is no evidence suggesting the same effect for semantically similar identifiers. In fact, the average debugging time is slightly lower for semantically similar identifiers.

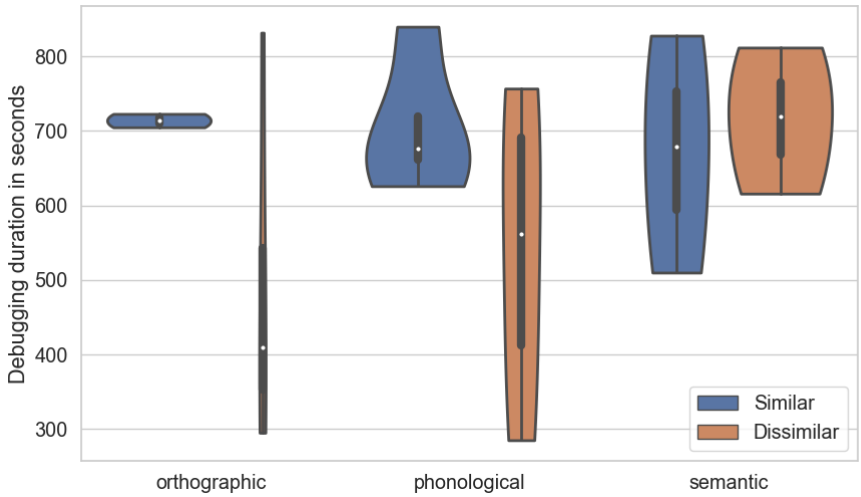

Figure 5 – Debugging time analysis in the pilot study.

On average, debugging code with orthographically similar identifiers takes 714 seconds, and debugging the same code with dissimilar identifiers takes 486 seconds. The difference is approximately 3.8 minutes, which may be substantial when scaled to bigger programs and projects. Nonetheless, there is no significant effect for orthographic similarity on debugging duration, t(10) = -1.27, p = .27, despite similar identifiers (M = 714, SD = 12) attaining higher debugging time than dissimilar identifier (M = 486, SD = 238).

Similarly, debugging code with phonologically similar identifiers on average takes 704 seconds, and dissimilar identifiers takes 541 seconds. Yet, there is no significant effect for phonologically similarity on debugging duration, t(10) = -1.4, p = .20, despite similar identifiers (M = 704, SD = 93) attaining higher debugging time than dissimilar identifier (M = 541, SD = 212).

The results of the pilot study called for further investigation of the influence of identifier similarity on code comprehension and debugging specifically. On one hand orthographic and phonological similarity seem to increase debugging time, and on the other hand similarity in semantics does not seem to have any effect on debugging time and success. Also, the differences between similar and dissimilar identifiers, although pronounced with orthographic and phonological similarity, does not appear to be statistically significant, possibly due to the small sample size. Therefore, we tried to address some of these issues in the main experiment, which we present next.

3. Experiment

In this section, we present the details of our main experiment which incorporated elements from the pilot study, in addition to new sources of data that we used to answer our research questions. Our controlled experiment consisted of 31 participants in total.

3.1. Objective

The pilot study provided no statistically significant evidence in support of the hypothesis that identifier similarity influences debugging success and time. Our goal in the main experiment was to improve our stimuli and incorporate additional sources of data to quantify the effect of identifier similarity on code comprehension and programmer workload.

- RQ1: Does identifier similarity influence success in debugging source code?
- RQ2: Does identifier similarity influence source code debugging time?
- RQ3: Does identifier similarity influence source code debugging difficulty?

With a larger sample size and an enhanced stimulus, we intended to provide better evidence to answer our research questions.

3.2. Participants

The 31 participants in our study were Computer Science students with experience ranging from one to eight years. Based on the k-means classification, 9 were classified as advanced programmer with four or more years of programming experience, and the remaining were classified as beginner. All participants had completed their second Computer Science course (CSII or equivalent) and were thus familiar with Java programming language, while some students had taken upper-level courses and had completed several internships as professional programmers. All participants were above the age of 18, and 17 were Female, 13 Male, and 1 Non-binary. Participation was voluntary, and participants were awarded a $15 gift card after the experiment.

3.3. Material

To enhance our experiment stimuli, we used a more systematic way of searching for similar identifier names in popular open-source projects. We used 10 popular Java repositories to mine for identifier names. We used repositories that fulfilled the guides of selecting meaningful repositories sets (Munaiah et al., 2017). The repositories represent approximately 3.9 million lines of code, after removing comments from code. Only Java files were processed; code in other languages was excluded. Table 1 shows the details of the selected Java repositories and the number of tokens in each repository.

Table 1 – Java repositories used to search for similar identifier names.

Repository	**Files**	**Lines**
Ant	1,314	304,957
Batik	1,651	353,516
Cassandra	2,673	586,451
Eclipse	154	25,914
Log4J	309	60,078
Lucene	8,467	1,874,373
Maven2	378	60,775
Maven3	834	113,384
Xalan-J	958	348,769
Xerces2	833	261,312
Total	17,571	3,979,251

From the repositories, we collected 170,823 unique identifiers. Using Wordnet (Miller, 1995) and a custom tool for calculating word-form overlap, we calculated the orthographic, phonological, and semantic similarity among the 1,000 most frequent identifiers in the code. These identifiers represent common identifier names that programmers use, and therefore they represent more realistic identifier names than the pilot study. From the resulting list of similar identifiers, the research team selected similar identifiers that could be integrated in realistic code snippets for use in our experiment.

```java
for(int count=0; count<100; count++){
        myDie.roll();
        int number = myDie.getSideUp();
        if ((number / 10) == 0) {
                results.add(myDie.getSideUp());
        }
```

(a) Similar

```java
for(int iteration=0; iteration<100; iteration++){
        myDie.roll();
        int sideUp = myDie.getSideUp();
        if ((sideUp / 10) == 0) {
                results.add(myDie.getSideUp());
        }
}
```

(b) Dissimilar

Figure 6 – Experiment semantic similarity material (identifiers highlighted).

We repeated the same structure as the pilot study with three programs and two versions of each program, one with similar and one with dissimilar identifiers. The orthographic similarity stimuli remained the same, since 'i' and 'j' are commonly used identifiers. For semantic similarity, we prepared a program with similar and dissimilar identifiers as shown in Figure 6. The semantically similar identifiers that we found in the Java repositories were 'count' and 'number' and the dissimilar identifiers were 'iteration' and 'sideUp'. Again, both versions included the same logical error.

```java
String right = stack.pop();
String operand = stack.pop();
String left = stack.pop();
write = left + " " + right + " " + operand;
stack.push(right);
```

(a) Similar

```java
String right = stack.pop();
String operand = stack.pop();
String left = stack.pop();
postfix = left + " " + right + " " + operand;
stack.push(right);
```

(b) Dissimilar

Figure 7 – Experiment phonological similarity material (identifiers highlighted).

For phonological similarity, we prepared a program that converted an infix expression to postfix. Figure 7 shows the two versions of the program with phonologically similar and dissimilar identifiers. The identifiers 'write' and 'right' were identical in pronunciation (homophones), while 'postfix' and 'right' were dissimilar.

3.4. Study Design

To answer our research questions, we collected three sources of data in our experiment: debugging time, success probability, and workload surveys. To answer our questions on the influence of identifier similarity on debugging success and time, we compared the debugging time and success probability of both similar and dissimilar versions of the same program, as we did in the pilot study. The order of the tasks was randomized, and the version of each program (similar/dissimilar) was also randomized.

Subjective Workload: To answer the research question focusing on the influence of identifier similarity on debugging difficulty and workload, we utilized NASA-Task Load Index (NASA-TLX) (Hart and Staveland, 1988). NASA-TLX has a wide range of applications (Grier, 2015) from aviation, physical activity, cognitive tasks, to software development (Fritz et al., 2014; Al Madi et al., 2022). The survey uses a multidimensional scale to represent six factors that contribute to workload and difficulty during various tasks. We used a simplified version of NASA-TLX which reported the magnitude of the experienced workload component on a scale from 0 to 100. The components are (Hart and Staveland, 1988):

- ***Mental Demand***: How much mental and perceptual activity was required?
- ***Physical Demand***: How much physical activity was required?

- ***Temporal Demand***: How much time pressure did you feel due to the pace at which the tasks or task elements occurred?
- ***Overall Performance***: How successful were you in performing the task?
- ***Effort***: How hard did you have to work (mentally and physically) to accomplish your level of performance?
- ***Frustration Level***: How irritated, stressed, and annoyed versus content, relaxed, and complacent did you feel during the task?

The magnitude of experienced components was reported on each scale in increments of 5, where zero represented the lowest magnitude and 100 represented the highest. The overall score described the perceived workload in performing the task and is often dubbed TLX. This final score had no unit nor an upper limit. We used NASA-TLX after each trial to measure the subjective workload experienced by participants when debugging source code with similar/dissimilar identifier naming.

3.5. Experiment Procedure

First, the participant was entered into the experiment lab, and the general procedure of the experiment was explained along with an online consent form. If the participant agreed to proceed with the experiment, the online form collected the participant's programming experience in years, and demographic information. The first debugging task was presented and timed from the start until the participant fixed the bug in the code or 15 minutes passed. The comments in the file informed programmers of the main functionality of the code, but no information of number of bugs were provided. During the task, participants were free to modify and run the codes. If the 15-minute limit was reached, the debugging task was considered unsuccessful. The debugging tasks were presented in randomized order, and the identifiers in each task were presented in either similar or dissimilar condition. After each debugging task, the participant filled an online version of NASA-TLX to report the workload experienced in that task.

3.6. Experiment Results

RQ1: Does identifier similarity influence success in debugging source code?

Here we attempt to provide more evidence on the influence of identifier similarity on debugging success. We approach this by examining the debugging success data of advanced programmers in contrast to data from beginners. Figure 8 shows debugging success probability for advanced programmers when presented with programs containing similar and dissimilar identifiers. The results demonstrate decreased success when programmers were presented with similar identifiers in orthography or phonology, with no difference in semantic similarity.

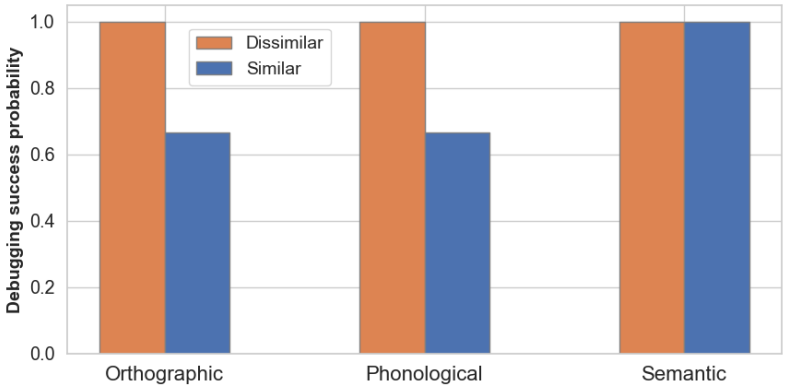

Figure 8 – Advanced programmers debugging success probability in the main experiment.

Looking at the data from beginners, we observe in Figure 9 the lower success probability in debugging for beginners, evident by the lower scores when compared to advanced programmers in figure 8. More

importantly, the differences between similar and dissimilar identifiers appear reversed in orthography and semantics.

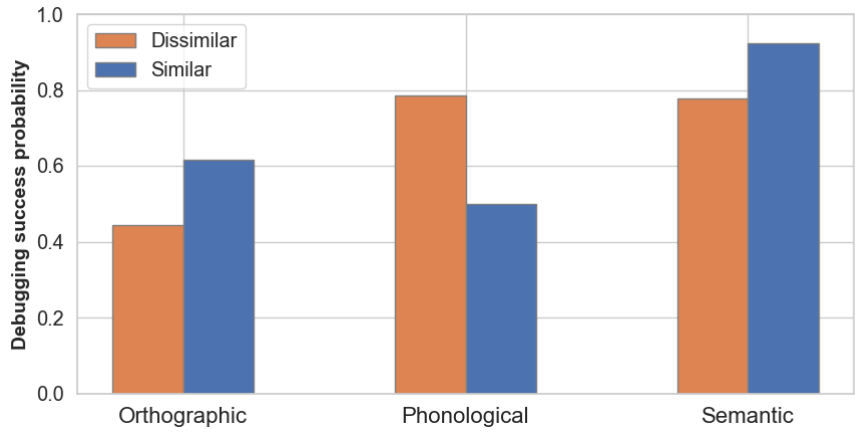

Figure 9 – Beginner programmers debugging success probability in the main experiment.

Comparing the results of beginners and advanced programmers, it appears that beginners have lower debugging success probability in general (comparing Figure 8 and Figure 9). Also, advanced programmers appear to be hindered by identifier similarity in orthography and phonology, while beginners only show reduced success with phonologically similar identifiers. Surprisingly, beginners seem to be advantaged by identifier similarity in orthography and semantics. Yet, none of the differences are statistically significant according to Fisher's test of independence. For advanced programmers Fisher's test results are p = .33 for orthography and p = .49 for phonology. For beginner programmers, Fisher's test results are p = .66 for orthography, p = .34 for phonology, and p = .54 for semantic similarity.

RQ2: Does identifier similarity influence source code debugging time?

Similar to the previous question, we examine the debugging duration of similar and dissimilar identifiers contrasting data from beginners to advanced programmers. Akin to the pilot study, we focus on the data from participants who successfully debugged the code.

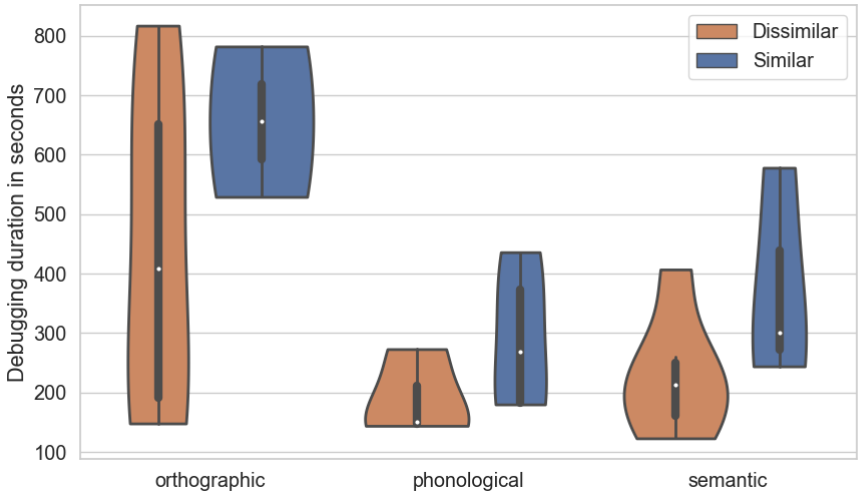

Figure 10 – Advanced programmers debugging time analysis in the main experiment.

Starting with Figure 10, which shows debugging duration for advanced programmers across similar and dissimilar programs, we notice a consistent pattern of similar identifiers leading to longer debugging time. This pattern is present in orthographic, phonological, and semantic similarity, and it is most

clear in orthographic similarity. On average, programmers takes 655 seconds to debug code with orthographically similar identifiers, and 439 seconds to debug the same code with dissimilar identifiers. The difference is approximately 3.6 minutes between similar and dissimilar identifiers. Nonetheless, there is no significant effect for orthographic similarity on debugging duration, t(7) = -0.98, p = .36, despite similar identifiers (M = 655, SD = 178) attaining higher debugging time than dissimilar identifier (M = 439, SD = 284).

For phonological similarity, programmers takes 288 seconds to debug code with similar identifiers, and 189 seconds to debug the same code with dissimilar identifiers. The difference is approximately 1.6 minutes between similar and dissimilar identifiers. Nonetheless, there is no significant effect for phonological similarity on debugging duration, t(7) = -1.19, p = .28, despite similar identifiers (M = 288, SD = 127) attaining higher debugging time than dissimilar identifier (M = 189, SD = 72).

For semantic similarity, programmers take 374 seconds to debug code with similar identifiers, and 227 seconds to debug the same code with dissimilar identifiers. The difference is approximately 2.45 minutes between similar and dissimilar identifiers. Nonetheless, there is no significant effect for semantic similarity on debugging duration, t(7) = -1.27, p = .14, despite similar identifiers (M = 374, SD = 178) attaining higher debugging time than dissimilar identifier (M = 277, SD = 101).

Examining the debugging duration data of beginners, Figure 11 presents a comparison between similar and dissimilar code debugging times. Unlike advanced programmers, the amount of time taken by beginners to debug programs with similar and dissimilar identifiers appears to be comparable, and in some instances code with dissimilar identifiers took longer to debug.

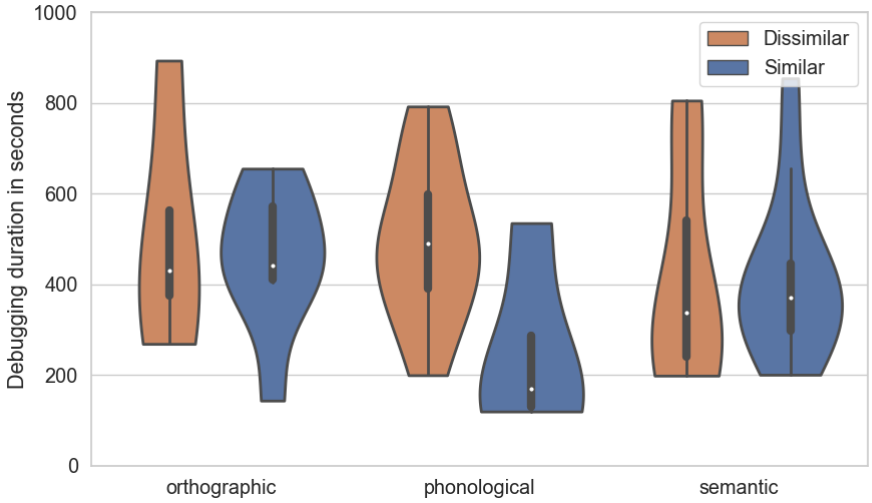

Figure 11 – Beginner programmers debugging time analysis in the main experiment.

For orthographic similarity, beginners take 627 seconds to debug code with similar identifiers, and 725 seconds for dissimilar identifiers. The difference is approximately 1.6 minutes, with a surprising advantage to similar identifiers. For phonological similarity, beginners takes 574 seconds to debug code with similar identifier, and 579 seconds for dissimilar identifier. The difference is 5 seconds on average. For semantic similarity, beginners takes 453 seconds to debug code with similar identifiers, and 522 seconds for dissimilar identifiers. The difference is approximately 1 minute, with an advantage to similar identifiers. None of the differences are statistically significant.

In summary, the results suggest that the influence of identifier similarity on debugging time is seen as an increase in debugging time in the data of experienced programmers, yet identifier similarity has no influence or reversed influence on beginner programmers. None of the differences appear to be significant on a statistical level.

RQ3: Does identifier similarity influence source code debugging difficulty?
To understand debugging difficulty, we measure participants' subjective workload via NASA-TLX. Figure 12 shows the Task Load Index (TLX) of advanced programmers in debugging similar and dissimilar programs. The results show increased workload and difficulty when programmers encounter similar identifiers in phonology, and lower workload debugging code with similar identifiers in orthography and semantics.

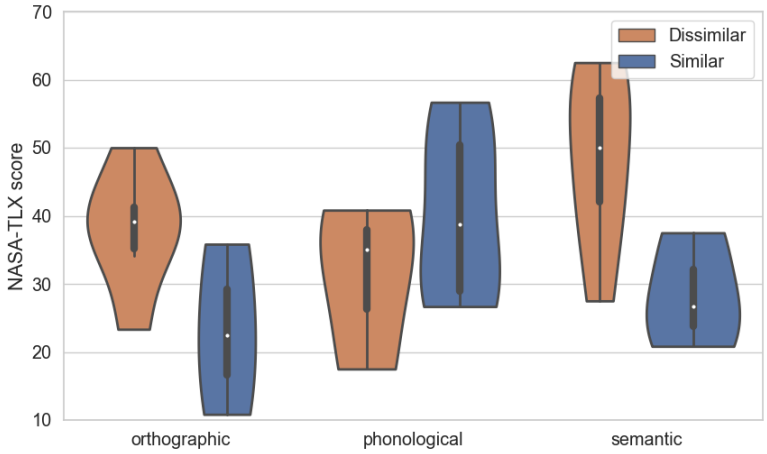

Figure 12 – Advanced programmers NASA-TLX in the main experiment.

For orthographic similarity, advanced programmers have an average NASA-TLX score of 23.1 (SD = 12.5) with similar identifiers and 38.0 (SD = 8.8) with dissimilar ones. Despite the huge difference of 14.9 with similar identifiers acquiring less workload than dissimilar identifiers, there is no significant effect for semantic similarity on NASA-TLX score, t(7) = 2.10, p = .07. For phonological similarity, advanced programmers have an average score of 40.1 (SD = 12.9) with similar identifiers, and 31.1 (SD = 12.1) with dissimilar identifiers. The difference between similar and dissimilar identifiers is approximately 9. In this case, similar identifiers increased debugging difficulty more than dissimilar identifiers. However, there is no significant effect for semantic similarity on NASA-TLX score, t(7) = -1.00, p = .34. Lastly for semantic similarity, advanced programmers have an average NASA-TLX score of 28.3 (SD = 8.5) with similar and 48.2 (SD = 5.3) with dissimilar ones. The gap goes to approximately 9.9 between similar and dissimilar identifiers, where similar identifiers make the debugging less challenging. However, we again fail to detect significant effect for semantic similarity on debugging duration as t(7) = -1.62, p = .15.

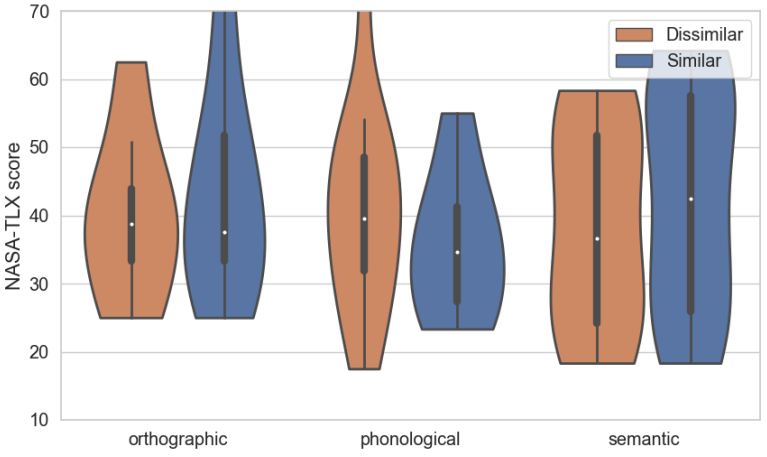

Figure 13 – Beginner programmers NASA-TLX in the main experiment.

On the other hand, Figure 13 illustrates the NASA-TLX data for beginners, where we observe no dif-

ference in the workload of debugging similar and dissimilar programs. For orthographic similarity, beginner programmers have an average NASA-TLX score of 42.7 (SD = 15.06) with similar identifiers and 40.0 (SD = 12.5) with dissimilar identifiers. Similar identifiers add slightly more difficulty to the debugging task as the mere difference of 2.7 indicates. For phonological similarity, beginner programmers have an average NASA-TLX score of 35.6 (SD = 10.8) with similar identifiers and 40.6 (SD = 14.8) with dissimilar identifiers. The difference is approximately 4 between similar and dissimilar identifiers, and debugging codes with similar identifiers is easier in this case. In addition, beginner programmers have an average NASA-TLX score of 42.6 (SD = 16.7) with similar identifiers and 38.6 with dissimilar ones (SD = 15.4) for semantic similarity. The difference is also 4, but this time programs with similar identifiers appear to be harder. Conducting statistical tests on the data, we find that there is no significant effect for all three types of similarity, with $t(19) = -0.44$, $p = .67$; $t(19) = 0.83$, $p = .42$; $U = 59.5$, $p = .97$ for orthographic, phonological, and semantic similarity respectively. Note that we use the Student T-test for orthographic and phonological similarity and the Mann-Whitney U Test for semantic similarity.

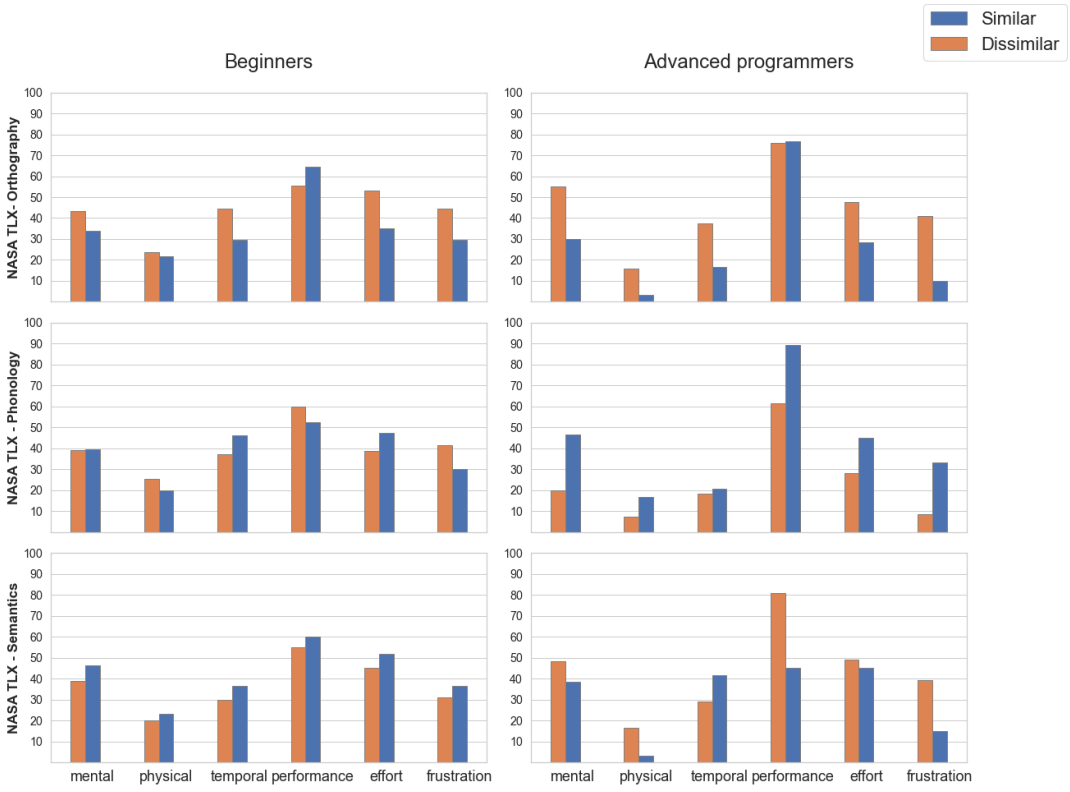

Figure 14 – Beginner programmers NASA-TLX in the main experiment.

Figure 14 further showcases breakdown scores of six components per task for both beginners and advanced programmers. All six graphs follow a general pattern: physical and temporal demand is less than mental demand, effort needed has a higher score than the frustration, and performance achieves the highest score among all six components. Intuitively comparing the NASA-TLX scores between beginners and advanced programmers, we state that experts experience less mental, physical, temporal demand, pay less effort to attain a higher performance in debugging, and meanwhile suffer from less frustration. However, after we conduct the Student T-test on the whole data, we argue that the difference is not a significant effect with $t(90) = -1.01$, $p = .32$, though advanced programmers report less overall workload ($M = 37.2$, $SD = 13.2$) than beginners ($M = 40.4$, $SD = 14.3$) do.

4. Discussion

Despite finding differences between similar and dissimilar identifiers with advanced programmers, the differences were not statistically significant, and therefore further research is needed to confirm/refute

that identifier similarity has an influence on debugging success, duration, or workload.

One of the takeaways from this paper is that comprehension is difficult to measure, and we use debugging as a "better" method of evaluating program comprehension when compared to multiple-choice questions or self-assessment. Nonetheless, it is possible that other factors, such as programming experience and debugging skill, play a bigger role in the overall success and performance in debugging compared to identifier naming. The previously mentioned study by Scanniello et al. (2017) highlights the significance of examining the different strategies programmers use when using debugging as an indicator of code comprehension.

Another important takeaway from this study is that only advanced programmers took longer and were less successful in debugging similar identifiers. Possibly, experience and repeated exposure to familiar programming structure such as 'i' and 'j' cause programmers to skip predictable code blocks where the error is located. Skipping predictable words is a well studied phenomena in natural language reading (Rayner, 1998). This possibility calls for a future experiment utilizing eye tracking to examine the skip probability of advanced and beginner programmers in relation to predictable and unpredictable identifier names.

An additional takeaway is that the three forms of similarity that we considered can potentially cause confusion in lexical access during reading. Lexical access describes the retrieval of word form, pronunciation, and meaning from memory during reading. Lexical access is usually measured on the level of millisecond (Rayner, 1998), and it is possible that the scale of our experiment (15 minutes) could mask the millisecond differences between similar and dissimilar identifiers. This idea, in combination with our results, calls for further investigation of similarity in identifier naming and its influence on code comprehension. Researchers might use a form of lexical decision task to accurately measure the difficulty or confusion that might occur on the millisecond level. It is also possible for eye tracking to shed light on hidden aspects of identifier similarity in reading code.

5. Conclusion and Future Work

Through a controlled human experiment (n=43), we explored the impact of orthographic, phonological, and semantic similarity on debugging success, time, and workload. We found that the observed differences in debugging success, duration, and workload were not statistically significant. These results and the details of the experiments we present in this paper call for further investigation of identifier similarity, with more focus on lexical access and potentially eye movement in code reading. Studying identifier similarity in code can shed light on new linguistic anti-patterns that could potentially hinder code comprehension.

Our future work includes extending several aspects of the current work. First, we would like to supplement the debugging task with other tasks to track programmers' true understanding of codes, such as code tracing and lexical decision tasks. In addition, we can explore other comprehension-impacting factors that we did not consider in this study. One example lies in our orthographic identifier pairs. While 'row' and 'col' are orthographically dissimilar, they convey more meaning than 'i' and 'j' as an orthographically similar pair. In addition, the physical distance between the identifiers in the code is another issue. It is possible that if a pair of identifier names is hundreds of lines apart, their similarity is unlikely to impact code understanding. Last but not least, we hope to collect more data from advanced and professional programmers and to affirm/refute the results of this study with more evidence.

References

Naser Al Madi, Siyuan Peng, and Tamsin Rogers. Assessing workload perception in introductory computer science projects using nasa-tlx. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 668–674, 2022.

Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. An investigation of compound variable names toward automated detection of confusing variable pairs. In *2021 36th*

IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pages 133–137. IEEE, 2021.

Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196. IEEE, 2013.

Boehm Barry et al. Software engineering economics. *New York*, 197, 1981.

Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2): 219–276, 2013.

Teresa Busjahn, Roman Bednarik, and Carsten Schulte. What influences dwell time during source code reading?: analysis of element type and frequency as factors. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 335–338. ACM, 2014.

Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE, 2015.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, 25(3):2140–2178, 2020.

Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th international conference on software engineering*, pages 402–413. ACM, 2014.

Rebecca A Grier. How high is high? a meta-analysis of nasa-tlx global workload scores. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 59, pages 1727–1731. SAGE Publications Sage CA: Los Angeles, CA, 2015.

Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988.

Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353. IEEE, 2007.

Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *14th IEEE international conference on program comprehension (ICPC'06)*, pages 3–12. IEEE, 2006.

Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *PPIG*, page 11. Citeseer, 2006.

Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–37, 2014.

Jonathan I Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 103–112. IEEE, 2001.

George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83, 2006.

Christian D Newman, Reem S AlSuhaibani, Michael J Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170:110740, 2020.

Keith Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.

Keith Rayner, Erik D Reichle, Michael J Stroud, Carrick C Williams, and Alexander Pollatsek. The effect of word frequency, word predictability, and font difficulty on the eye movements of young and older readers. *Psychology and aging*, 21(3):448, 2006.

Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. Fixing faults in c and java source code: Abbreviated vs. full-word identifier names. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(2):1–43, 2017.

Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports*, pages 65–86. ACM, 2010.

Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3): 219–238, 1979.

Janet Siegmund and Jana Schumann. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, 20(4):1159–1192, 2015.

Thomas A Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, (5): 494–497, 1984.

Armstrong A Takang, Penny A Grubb, and Robert D Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

Rebecca Tiarks. What maintenance programmers really do: An observational study. In *Workshop on Software Reengineering*, pages 36–37. Citeseer, 2011.

Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

Anneliese Von Mayrhauser, A Marie Vans, and Adele E Howe. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, 9(5): 299–327, 1997.