Descriptive Compound Identifier Names Improve Source Code Comprehension

Andrea Schankin
Karlsruhe Institute of Technology
Karlsruhe, Germany
schankin@teco.edu

Annika Berger
Karlsruhe Institute of Technology
Karlsruhe, Germany
berger@teco.edu

Daniel V. Holt
Heidelberg University
Heidelberg, Germany
daniel.holt@uni-heidelberg.de

Johannes C. Hofmeister
University of Passau
Passau, Germany
johannes.hofmeister@uni-passau.de

Till Riedel
Karlsruhe Institute of Technology
Karlsruhe, Germany
riedel@teco.edu

Michael Beigl
Karlsruhe Institute of Technology
Karlsruhe, Germeny
michael.beigl@kit.edu

ABSTRACT

Reading and understanding source code is a major task in software development. Code comprehension depends on the quality of code, which is impacted by code structure and identifier naming. In this paper we empirically investigated whether longer but more descriptive identifier names improve code comprehension compared to short names, as they represent useful information in more detail. In a web-based study 88 Java developers were asked to locate a semantic defect in source code snippets. With descriptive identifier names, developers spent more time in the lines of code before the actual defect occurred and changed their reading direction less often, finding the semantic defect about 14% faster than with shorter but less descriptive identifier names. These effects disappeared when developers searched for a syntax error, i.e., when no in-depth understanding of the code was required. Interestingly, the style of identifier names had a clear impact on program comprehension for more experienced developers but not for less experienced developers.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**; • **Software and its engineering →** *Software usability*; *Error handling and recovery*; *Maintaining software*;

KEYWORDS

Program Comprehension, Identifier Names, Java Developers, Software Quality

ACM Reference Format:
Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive Compound Identifier Names Improve Source Code Comprehension. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3196321.3196332

1 INTRODUCTION

Developing software requires comprehending source code [9], whether it is developing new software, maintaining existing software or integrating new functionalities [24]. When improving software, the integration of the actual function may need less time than understanding the style and intention of the initial software engineer [16]. Investigating how developers understand source code may allow us to improve code comprehensibility, leading to reduced time and costs.

Code comprehension depends on the quality of code, which is impacted by program features such as code structure, comments, and identifier naming [19]. Naming conventions are usually provided to improve consistency within and across projects. While some guidelines are aimed at increasing readability, e.g using camelCase [6, 7, 25], other naming conventions allow differentiating identifiers with regard to their functionality within the code. However, these conventions do not necessarily lead to better code comprehension because the actual meaning or function of the code may not have been expressed clearly [14]. This idea has been addressed by approaches such as *clean code* [22] or *domain-driven design* [15], which are supposed to allow intuitive code comprehension, without much effort in short time.

However, *empirical evidence* with regard to the appropriate length of identifier names is still sparse. It has been shown that very short identifier names often do not convey information in sufficient detail [17, 19]. They are as impractical as excessively long identifiers, which may overload working memory [20].

In this paper we describe an experimental study in which we quantify the impact of identifier naming style on program comprehension. Psychological research on comprehensibility of natural-language texts supports both short and long identifier names. On the one side, the *word-length effect* [5] would predict that short identifier names improve code comprehension because they require less cognitive capacity than longer ones. This effect describes that lists of short strings are easier to remember than lists of long strings. Thus, the performance of developers may be better with short identifier names because more items fit into working memory, allowing a better overview of the code. In contrast, a word's meaning can help to relieve cognitive resources, either through a process called

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196332

chunking [5], in which items are regrouped to more meaningful units, or by *activating concepts in long-term memory.* Both processes predict a better comprehension performance for longer but more descriptive identifier names.

We make three contributions:

(1) We show that code containing descriptive compound identifier names is comprehended faster than those with short single-word identifiers in Java. Thus, we replicate and extend the findings by Lawrie et al. [19, 20] and Hofmeister et al. [17], who have reported improved source code comprehension when identifiers were named with words instead of single letters or abbreviations.
(2) We improve the ecological validity of findings by using more complex code snippets, consisting of a class with one to three methods. We used identifier names that may be used in real software development projects.
(3) We explore the impact of programming expertise on code comprehension. Our results show that experienced programmers clearly benefit from descriptive compound identifier names, whereas novices do not.

2 RELATED WORK

Since the length of identifier names is no longer limited by compilers, programmers have great freedom to select names that promote code understanding. Well chosen longer identifier names with embedded subwords (i.e., compound identifier names) can carry more information than single-word or even single-letter identifiers [21], e.g., `notConverted` compared to `input` as parameter of a function that converts a string from camelcase to underscore. However, what is the impact of descriptive compound identifier names on code comprehension?

Code comprehension has been measured in many different ways tapping various cognitive processes (e.g., perception, attention, memory, and reasoning). We will briefly summarize related work for different tasks and measures of code comprehension.

Semantic description of the function. Lawrie et al. [19] varied the identifier length between single letter, abbreviation and full word. In the latter two conditions names could consist of a single word or a compound. As a measure of program comprehension, participants were asked to describe the functionality of a code snippet in their own words and their confidence in the description. Program comprehension was better for abbreviated and word identifier names in comparison to a single letter.

Defect detection. In a similar study by Hofmeister et al. [17], participants' time to find an error in the code was measured. The authors varied the length of identifier names between single letter, abbreviation and word. In contrast to the study by Lawrie et al. [19], single words but no compounds were used. Program comprehension was better for full words compared to single letters and abbreviations, but only if an in-depth understanding of the code was required.

Remember and recall. Binkley et al. [8] investigated the length of identifier names on short- and long-term memory. To name a method, the authors used Java chain expressions (i.e., a chaining of method calls and field selectors) of different length. Participants were asked to remember the name and to recall one part of it after having worked on a distracting task. Although participants needed longer to remember names with increasing length, there was no negative effect on correct recall for experienced Java developers if the name could be tied to concepts in long-term memory.

Acceptance. Relf [23] measured the subjective acceptance of 19 naming guidelines, amongst others using short names (i.e., identifiers shorter than 8 characters), long names (i.e., identifier names longer than 20 characters), and compound names (i.e., identifiers should consist of two to four words). Expert programmers accepted these three guidelines, even if they were cognitively more demanding.

Code quality. Butler et al. [10] investigated the relationship between the quality of identifier names and code quality (measured with the code analysis tool FindBugs[1]) in eight established open source Java applications. The quality of code was best with identifier names composed of two to four words. In contrast, Aman [1] reported that Java methods collected from nine popular open source products were more fault-prone when local variables were named with compounds. To measure code quality, the authors counted the number of bug fixes, assuming that a smaller number were related to a better code quality. However, one may argue that bugs may simply not have been found in methods with short identifier naming.

3 RESEARCH QUESTION AND HYPOTHESES

In summary, previous research has shown that longer descriptive identifier names may improve code comprehension. However, to our knowledge no empirical study has directly compared short identifiers consisting mostly of single words and longer but more informative compound identifiers yet. Before motivating our hypotheses, we introduce the measurement we used as well as our experimental rationale.

3.1 Measuring Code Comprehension (Dependent Variables)

Task completion time. We operationalized the performance of comprehension by measuring how long developers investigated a snippet of code to find a defect. The response is easy to score for *correctness* and it has a well-defined time point, allowing reaction time analyses. Locating defects is a common programming task, which renders it a relevant target for studying program comprehension. We assumed that *semantic* defects in code can only be found and corrected when it is understood, because developers cannot evaluate the consequences of their changes otherwise (see [17]).

Visual focus. In order to explore the actual process of reading source code a restricted focus viewer was used [18]. The viewer allowed us to record how much time developers spent looking at different parts of the code and to observe their direction of reading.

3.2 Experimental Variation (Independent Variables)

Identifier naming style. In our study we compared short, mostly single-word identifier names ("short" condition) with longer compound identifier names ("compound" condition). It is important

[1] https://findbugs.sourceforge.net/

to note that the compound identifiers we created did not simply consist of more characters but were also more descriptive, carrying more information about the intention of a function. Because the scope of the variables and methods was rather small in the code snippets we used, short identifiers should be sufficient to describe the function [1]. However, we propose that longer but more descriptive compound identifier names allow a better mapping to the mental model of the software engineer about the problem domain. This may have two effects. First, identifiers might be easier and faster to remember and to recall from long-term memory. Second, inconsistencies between the code and the mental model might lead to a faster detection of semantic defects.

Type of defect. Finding *semantic defects* requires that the intentions behind the code (what should it do?) and the semantics of its operation (what does it do?) are understood to give a correct response. The code compiles without errors but the result of the method is not as intended. As a control condition we tested whether identifier naming style affects performance in tasks in which no in-depth understanding of the code is required. This allowed us to evaluate whether the naming conditions selectively interfered with high-level program comprehension or whether they also affect lower-level reading processes. To accomplish this, we measured how much time developers needed to locate a *syntax error*. Syntax errors, such as missing brackets or semicolons, render the code invalid but require no in-depth understanding of identifier names' meanings to be corrected.

3.3 Hypotheses

The objective of our empirical study was to investigate the impact of descriptive compound identifier names on code comprehension. If more descriptive identifier names improve code comprehension, we expect an effect on the time required to find an error in the source code (i.e., task completion time), but only when searching for a semantic defect. Previous research has shown that programming experience might moderate the impact of identifier naming on code comprehension [19, 23]. However, the direction of the effect is unclear. In summary, we tested the following hypotheses:

> $H_{semantic}$: *Semantic defects in the source code are detected faster with longer but more descriptive identifier names.*
>
> $H_{syntactic}$: *The length of the identifier name has no effect on the time required to find syntax errors in the source code.*
>
> $H_{experience}$: *Programming experience moderates the impact of identifier length on code comprehension (exploratory).*

4 EXPERIMENTAL SETUP

The hypotheses were tested in a web-based experimental study. An overview of the experiment is given in Table 1.

4.1 Source Code Selection

We created four code snippets in Java to have full control over the stimulus materials and to ensure that no participant had seen the materials before. All code snippets contained algorithms which were simple enough to be comprehensible in a reasonable time frame but complex enough for defects to "hide" in the code. In a small pre-study with five participants we tested whether it was possible to find each defect within 10 minutes and whether snippets were of comparable difficulty. Each of the four snippets consisted of a class with between one and three methods and comprised 27 lines of code, i.e., the scope of variables and methods was kept small.

We then created four variants of each snippet, varying identifier naming and task type. Figure 1 shows an example of the two naming variants of a function. In Table 2 a direct comparison of descriptive compound and short single-word names of the function "converter" is displayed. For the *short* condition, we chose identifier names that described the function of the method, parameter, or variable as precisely as possible, mainly with one word. Longer identifier names were created by choosing names that described the function of the method, parameter, or variable in more detail. These identifiers were mainly named with compounds of two or three words. In each of the two naming variants an error was placed, either a semantic defect or a syntax error. The errors were placed in similar locations in the code to avoid a bias caused by the location. Table 3 summarizes the experimental conditions and their mapping to the hypotheses. All code snippets were presented with common syntax highlighting.

Table 1: Experiment overview

Goal	Study the impact of identifier names on program comprehension
Independent Variables	Identifier naming (short single-word vs. long descriptive compound), Task (semantic defect, syntax error)
Task	Identify semantic defect
Control	Identify syntax error
Dependent Variable	Time to find defect
Secondary Measures	Visual attention, Correctness
Potential Confounding Factors	Materials, Inter-individual differences, Item order
Design	Within-subjects

Table 2: Examples of the different identifier naming styles

Type	Descriptive compound identifier	Short identifier
Class	UnderScoreCamelCaseConverter	NotationConverter
Method	convertUnderScoreToCamelCase	camelcase
Parameter	underScore	input
Variable	singleWordParts	parts
Variable	convertedInput	result
Method	capitalizeFirstLetter	capitalize
Parameter	notConverted	input
Variable	firstLetter	first
Variable	otherLetters	other
Variable	convertedWord	converted
Method	lowercaseFirstLetter	lowercase

Short Variant

```
1  // Offers methods to convert text between camelcase (e.g., textInNotation)
2  // and underscore (eg., text_in_notation) notation
3  public class NotationConverter {
4
5     public String camelcase(String input) {
6        String[] parts = input.split("_");
7        String result = parts[0];
8        for (int i = 1; i < parts.length; i++) {
9           result += lowercase(parts[i]);
10       }
11       return result;
12    }
13
14    private String capitalize(String input) {
15       String first = input.substring(0, 1);
16       String other = input.substring(1);
17       String converted = first.toUpperCase().concat(other);
18       return converted;
19    }
20
21    private String lowercase(String input) {
22       String first = input.substring(0, 1);
23       String other = input.substring(1);
24       String converted = first.toLowerCase().concat(other);
25       return converted;
26    }
27 }
```

Descriptive Compound Variant

```
1  // Offers methods to convert text between camelcase (e.g., textInNotation)
2  // and underscore (e.g., text_in_notation) notation
3  public class UnderScoreCamelCaseConverter {
4
5     public String convertUnderScoreToCamelCase(String underScore) {
6        String[] singleWordParts = underScore.split("_");
7        String convertedInput = singleWordParts[0];
8        for (int i = 1; i < singleWordParts.length; i++) {
9           convertedInput += lowercaseFirstLetter(singleWordParts[i]);
10       }
11       return convertedInput;
12    }
13
14    private String capitalizeFirstLetter(String notConverted) {
15       String firstLetter = notConverted.substring(0, 1);
16       String otherLetters = notConverted.substring(1);
17       String convertedWord = firstLetter.toUpperCase().concat(otherLetters);
18       return convertedWord;
19    }
20
21    private String lowercaseFirstLetter(String notConverted) {
22       String firstLetter = notConverted.substring(0, 1);
23       String otherLetters = notConverted.substring(1);
24       String convertedWord = firstLetter.toLowerCase().concat(otherLetters);
25       return convertedWord;
26    }
27 }
```

Figure 1: The two naming variants of the function that converts text between camelcase and underscore. This snippet contains a semantic defect in line 9, *lowercase* should be *capitalize.*

4.2 Data Collection

The experiment was administered over the web. The application we used was originally developed by J. Hofmeister [17] and was adapted for the present study. For example, we included individual performance feedback to motivate developers to participate. The code is available on GitHub[2].

After answering some general demographic questions, participants completed an example problem before starting to work on the actual code snippets. Participants were asked to find one defect in a snippet of code, which was repeated four times. After they had completed two snippets with a semantic defect, participants were asked to complete two more snippets but looking for a syntax error.

When participants had found the defect, they pressed the space bar, which froze the viewer window and opened a dialog screen. Here they entered the line number of the defect, a description, and a correction (see Figure 2 right). Participants who had failed to find the defect in a snippet after three attempts were allowed to finish the experiment, but their data was excluded.

To gather coarse-grained data about participants' visual focus, we used an implementation of the restricted focus viewer [18], which limited participants' view on the code to seven lines at a time (approximately one forth of the complete snippet). The viewer window could be shifted up and down a line by using the arrow keys to reveal different parts of the code (see Figure 2 left).

Table 3: Experimental conditions and hypotheses

		(iii) Identifier Length	
(i) Code Snippet	(ii) Type of Error	Short	Compound
converter	semantic	H_{sem}	H_{sem}
	syntactic	H_{syn}	H_{syn}
emailuser	semantic	H_{sem}	H_{sem}
	syntactic	H_{syn}	H_{syn}
familymember	semantic	H_{sem}	H_{sem}
	syntactic	H_{syn}	H_{syn}
linkedlist	semantic	H_{sem}	H_{sem}
	syntactic	H_{syn}	H_{syn}

4.3 Experimental Design

We performed a web-based experimental study, in which we varied the style of the identifier name (short, mostly single word vs. long, descriptive compound name) and type of error (semantic vs. syntactic) as independent variable in a within-subjects design. With this design, we controlled for inter-individual differences between participants.

All participants started with two semantic defects followed by two syntax errors. The order of identifier naming styles (short vs. long identifier names) were counterbalanced to control for order effects. For each participant one of these sequences was randomly selected and combined with a random ordering of the four source code snippets.

In summary, every participant saw:

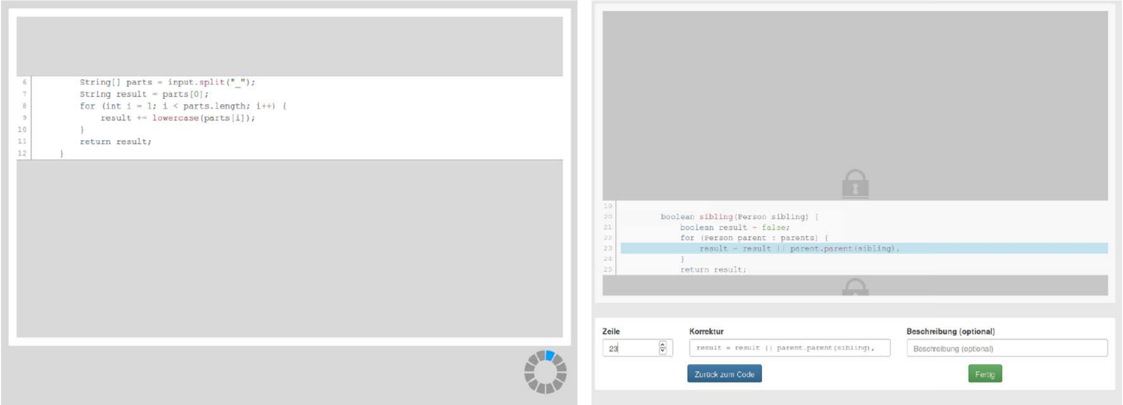
- all four snippets, encountering each snippet only once
- two semantic defects first, then two syntax errors
- both identifier naming styles, one for each type of error

4.4 Data Preparation and Analysis

Task completion time. We measured how long participants looked at the code until they indicated that they had found the defect by pressing the space bar. We subtracted the time spent answering the dialog and only evaluated the time that participants interacted with code.

Accuracy. Participants indicated the location of an error in the code by typing in the line number and correcting the error. Number of fails were used to measure accuracy.

[2] https://github.com/acBerger/peter



```
        String[] parts = input.split("_");
        String result = parts[0];
        for (int i = 1; i < parts.length; i++) {
            result += lowercase(parts[i]);
        }
        return result;
    }
```

```

    boolean sibling(Person sibling) {
        boolean result = false;
        for (Person parent : parents) {
            result = result || parent.parent(sibling),
        }
        return result;
```

Zeile: 23

Korrektur: result = result || parent.parent(sibling),

Beschreibung (optional): Beschreibung (optional)

Figure 2: Screen shot of the UI. Participants saw only parts of the code snippet in a window they could move (left). After pressing the space bar to indicate that they had found the defect, participants entered the line number, a short description, and a correction.

Focus of Attention. By presenting only 7 lines of source code in a sliding window, we were able to observe participants' focus of attention during reading. For data analyses, the source code was divided into four areas of interest (AOIs) (cf. [17]).

AOI$_{\text{Comment}}$ comprised all comment lines at the top of the snippet, AOI$_{\text{Pre-Defect}}$ comprised the lines of code before the defect becomes visible in the letterbox, AOI$_{\text{Defect}}$ comprised the lines where the defect is visible in the letterbox, and AOI$_{\text{Defect}}$ comprised all lines after the defect until the bottom of the snippet. Each letterbox position was attributed to the AOI which contained the center line of the letterbox. For our analyses we evaluated the AOIs' dwell times and visits. The times spent within an AOI were summed as the AOI's dwell time. Each movement from an AOI to another was marked as a visit.

Changes in reading direction. A change in reading direction was counted when the sliding window was moved in the opposite direction for at least two consecutive moves.

4.5 Participants Demographics

Participants were invited to our experiment via direct contact and a student e-mail list at the Karlsruhe Institute of Technology, Germany. Overall, 187 students and professional developers started to participate in the online experiment and 148 completed it. To improve data quality, the following selection criteria were applied (cf. Table 4). One participant with insufficient Java skills was excluded as were participants who did not complete all tasks, needed more than 3 attempts or more than 10 minutes to succeed in finding a defect. Furthermore, participants who reported encountering a distraction or not having worked conscientiously were excluded. Data from two participants with unusually short response times were retained in the analysis as they did not have a strong effect on overall results.

In total, 88 participants (82 male, 3 female, 3 did not indicate their gender) remained in the sample, aged between 19 and 32 years (median: 23 years). Most of them had a bachelors ($N = 48$) or masters ($N = 7$) degree in computer science, other started to study ($N = 27$). The overall programming experience ranged between 1 and 16 years (median: 6 years) in 5 programming languages on average. The experience with Java ranged between less than a year and 10 years (median: 3 years).

Table 4: Exclusion criteria

		Criterion	n
Language Level	German(1-6)	<3	0
	English(1-6)	<2	0
Experience	Java Skill Level (0-4)	<1	1
Behavior	Completed all tasks?	No	39
	Encountered distraction?	Yes	23
	Worked conscientiously?	No	4
	Attempts to succeed	>3	26
	Too slow (time per trial)	>10min	28
Other	Participated before?	Yes	4
	Total (Criteria not mutually exclusive)		99

5 EXPERIMENTAL RESULTS

In this section we report the results with regard to the hypotheses. We calculated inferential statistics for semantic defects and syntax errors separately. We employed a significance level of $\alpha = 0.05$ for all tests.

To facilitate evaluating the practical significance of results, we calculated standardized effect sizes. For within-subjects differences, we report Cohen's d_z with a correction for correlated observations [12]. The proportion of variance explained (R^2) is another commonly used index of effect size, which we report for some of the central results. For interpretation we followed the conventions suggested by Cohen [12] (see Table 5).

5.1 Semantic Defects (H$_{\text{semantic}}$)

To assess the impact of identifier names on the time required to find a semantic defect, a paired t-test was run with identifier style (short

Table 5: Conventions for the interpretation of effect sizes [12]

Interpretation	Cohen's d_z	R^2
small effect	0.20	0.01
medium effect	0.50	0.06
large effect	0.80	0.14

single-word vs. long descriptive compound) as independent variable and task completion time as dependent variable. Participants were about 29.5 seconds (or 14%) faster in finding a semantic defect when identifiers were named with descriptive compounds compared to short single words. This difference was statistically significant, $t(87) = 2.005, p = 0.048$, but small ($d_z = 0.27$). It explains about 2% of the variability in the data ($R^2 = 0.02$). The effect is illustrated in Figure 3 (left). The number of fails to report the error was not affected by the style of the identifier names, $t(87) = 0.427, p = 0.671, d_z = 0.09$.

The analysis of the AOIs reflects the visual focus of the participants during code reading (see Figure 4 left). The identifiers' naming style had no impact on the amount of time participants spent with reading the commentaries at the beginning of the code snippets, $t(87) = 0.961, p = 0.339, d_z = 0.14$, or at the defect itself, $t(87) = 1.137, p = 0.258, d_z = 0.16$. However, with compound identifier names, participants focused their attention significantly longer on the part before the defect, $t(87) = 2.566, p = 0.012, d_z = 0.39$, whereas they spend more time after the defect with single-word identifier names, $t(87) = 2.286, p = 0.025, d_z = 0.32$. Again, these effects were of small size, explaining about 2-3% of the variability in the data.

The identifiers' naming style also affected the number of changes in reading direction (see Figure 5 left). Participants changed the direction more often, i.e. they scrolled back and forth more frequently, when identifiers were named with single words, $t(87) = 2.968, p = 0.004, d_z = 0.37$. This is a small effect, which explains about 4% of the variability in the data ($R^2 = 0.04$).

In summary, the style of the identifier names affected how source code was read. With descriptive compound identifier names, participants spent about 7% more time in the lines of code *before* the actual defect occurred. They changed 24% less often their reading direction and were able to find the semantic defect about 14% faster than with short single-word identifier names.

5.2 Syntax Errors ($H_{syntactic}$)

As a control condition, participants were asked to find syntax errors. To complete this task, only knowledge about the syntax but not about the content of a function is necessary. Therefore, we predicted to observe no impact of identifier naming style on the dependent variables. A power analysis indicated that the sample size was sufficient to detect effects even of small-to-medium size (Cohen's $d_z = 0.3$) with a probability of 80%, i.e., finding a non-significant result suggests that the effect is at most relatively small.

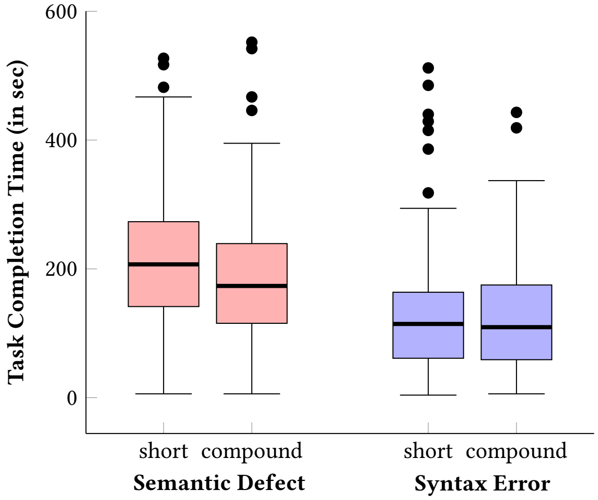


Figure 3: Task completion times for finding semantic defects (left) and syntax errors (right) in the source code, separately for short and descriptive compound identifier names.

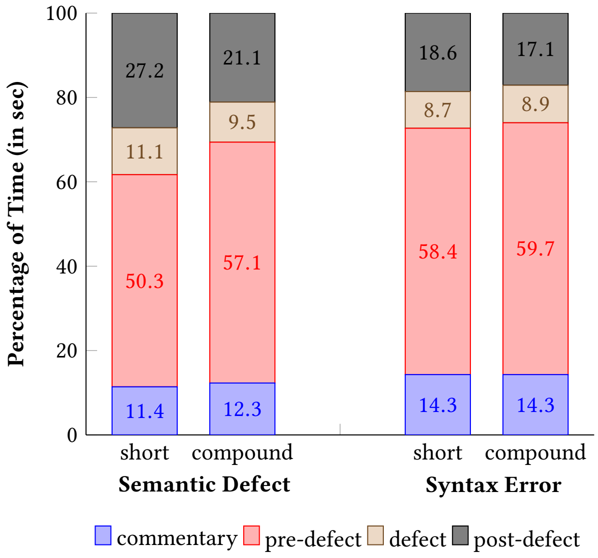


Figure 4: Distribution of task completion time onto different areas of interest (AOIs)

To assess the impact of identifier names on the time required to find syntax errors, a paired t-test was run with identifier style (short single-word vs. long descriptive compound) as independent variable and task completion time as dependent variable. As expected, there was no significant difference, $t(87) < 0.653, p = 0.515, d_z = 0.09$. The effect is illustrated in Figure 3 (right). The number of fails to report the error was not affected by the style of the identifier names, $t(87) = 0.352, p = 0.726, d_z = 0.07$.

The visual focus during code reading was assessed by comparing the time spent in different AOIs (i.e., commentary, pre-defect, defect, and post-defect; see Figure 4 right). This was not affected by the identifiers' naming styles when looking for a syntax error in the code, i.e., there was no significant difference between single

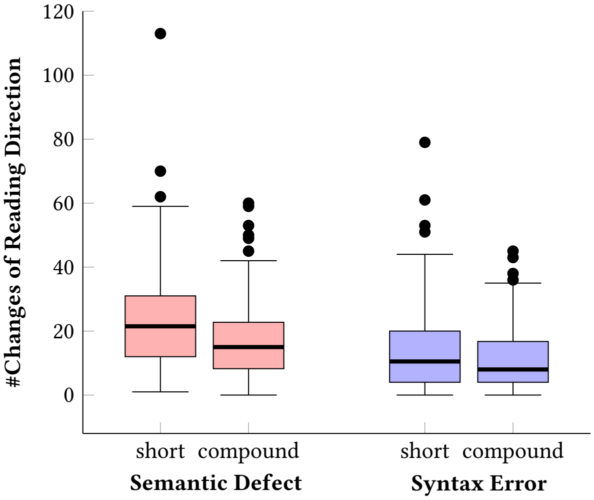


Figure 5: Number of changes in reading direction while finding a semantic defect (left) or a syntax error (right), separately for short and descriptive compound identifier names.

and compound identifier names in the reading time for different parts of the code snippets, all t's$(87) < 1, p$'s$> 0.563, d_z < 0.09$. Furthermore, the reading direction was not affected by the naming style of the identifiers, $t(87) = 1.550, p = 0.125, d_z = 0.21$ (see Figure 5 right).

In summary, the style of the identifier names had no effect on finding a syntax error in the source code, neither on how the code was read nor in the time required to find the error.

5.3 Exploration: Impact of Programming Experience ($H_{\textbf{experience}}$)

Based on their overall programming experience in any programming language (in years), participants were divided into two groups by median split. Novice programmers ($N = 45$) were between 19 and 26 years old (median: 22 years), have had programmed 1 to 6 years (median: 5 years), with 4 different programming languages (range: 1-13). Their experience with Java ranged between less than a year and 6 years (median: 3 years). Experienced programmers ($N = 43$) were between 20 and 32 years old (median: 24 years), had between 7 and 16 years (median: 8 years) of experience in programming with 6 different programming languages (range: 3-22). Their experience with Java ranged between less than a year and 10 years (median: 5 years).

With the following analyses, we explored whether the impact of identifier names on finding *semantic* defects in the source code depends on programming experience. The descriptive statistics are presented in Table 6. We calculated t-tests separately for each group (i.e., novice and experienced developers) to estimate the effect of identifier names in each group. It should be noted that although this analysis allows conclusions about the impact of identifier names in each group, it is not possible to assess whether the impact differs between the groups.

Novice developers. For novice developers, identifier names did neither affect significantly task completion time, $t(44) = 0.532, p = 0.597, d_z = 0.10$, the number of fails to report the error, $t(44) = 1.062, p = 0.294, d_z = 0.22$, nor the time spent in different parts of the source code (i.e., commentary, pre-, at, or post-defect), all $t(44)$'s$< 1.876, p$'s> 0.066. The largest, but still very small, effect was observed for the time spent after the defect, $t(44) = 1.875, p = 0.067, d_z = 0.12$, which explains about 1% of the variability of the data ($R^2 = 0.01$). The number of changes in reading direction was not affected by identifier names, $t(44) = 0.110, p = 0.913, d_z = 0.02$.

Experienced developers. In contrast to novices, more experienced developers were much faster in finding a semantic defect with compound identifier names, $t(43) = 2.724, p = 0.009, d_z = 0.52$. This moderately strong effect explains about 6% of the variability in the data ($R^2 = 0.06$). The number of fails did not differ between single and compound identifier names, $t(42) = 0.724, p = 0.473, d_z = 0.15$. The visual focus of experienced programmers varied with identifier naming style. When compound identifier names were used, experienced programmers spent more time at the lines before the defect occurred, $t(42) = 2.362, p = 0.023, d_z = 0.49$, explaining 6% of the variability in the data ($R^2 = 0.06$). There were no effects for other parts of the source code, all t's$(42) < 1.309, p$'s$> 0.197, d_z = 0.26$. Possibly compound identifier names were remembered better. This hypothesis was supported by the finding of fewer changes in reading direction with compound identifier names, $t(42) = 2.419, p = 0.020, d_z = 0.45$, which explains about 5% of variability in the data ($R^2 = 0.05$).

In summary, the style of identifier names had a moderate impact on program comprehension for more experienced developers but only a minor impact on how novice developers read or comprehended source code. The reported findings are not only statistically significant but their medium effect sizes also point to their practical significance.

6 DISCUSSION

The objective of the study was to assess the impact of identifier length on code comprehension, given that longer identifiers are more informative and not simply longer in the number of characters.

Our study shows that longer but more informative identifier names improved code comprehension when an in-depth understanding of the code was required, i.e., to find and correct a *semantic defect.* Participants needed about 14% less time with compound identifier names than with shorter single-word identifiers. Also, they read the code more serially, jumping less frequently back and forth. A more detailed analysis showed that these effects depended on programming experience. Experienced but not novice programmers significantly benefited from longer, descriptive identifier names. If no detailed understanding of the code was necessary, i.e., to find and correct *syntax errors*, identifier length had no impact on comprehension.

Previous studies have indicated that longer, more informative identifier names improve code comprehension [17, 19, 20]. Our study extends to results by Hofmeister et al. [17] who tested single letters, abbreviations, and single words with a clear advantage for words. Using the same measure of code comprehension, our study

Table 6: Summary of descriptive statistics.

		Novice		Expert	
Measure	**Style**	*M*	*SD*	*M*	*SD*
Task completion	Single	238.1	116.7	207.7	109.7
	Compound	225.6	127.3	160.4	63.3
Fails	Single	0.2	0.5	0.2	0.5
	Compound	0.4	0.7	0.1	0.5
Comment	Single	11.4	5.8	11.4	8.0
	Compound	13.6	7.5	11.0	5.3
Pre-Defect	Single	48.9	20.8	51.8	16.3
	Compound	54.7	1.2	59.6	16.0
Defect	Single	9.3	9.1	12.9	11.4
	Compound	8.8	9.2	10.2	9.1
Post-Defect	Single	30.4	19.7	23.9	16.2
	Compound	22.9	16.6	19.3	14.1
Reading Direction	Single	14.6	16.0	15.1	14.2
	Compound	14.3	10.8	9.6	9.6

shows that even longer names furhter improve the speed of code comprehension.

In the following we will discuss our results with regard to underlying cognitive processes and programming expertise. We will then describe threats to validity, before concluding with practical implications.

6.1 Cognitive Processes

Code comprehension involves various cognitive processes, e.g., perception, attention, memory, and reasoning. Because the length and meaning of identifiers is closely related to working memory, we will discuss its role in more detail.

Working memory is responsible for temporarily holding information available for conscious processing. We use it, for example, for reasoning or decision-making. New information can be stored for about 15 to 30 seconds without rehearsal (i.e., repetition). According to the widely used model by Baddeley [3], working memory consists of thee subsystems, a *phonological loop* to store auditory information, a *visual-spatial sketchpad* for visual information, and an *episodic buffer* that links information across domains to form integrated units of visual, spatial, and verbal information with time sequencing. The episodic buffer is also assumed to have links to long-term memory and semantic meaning [2]. A forth component, the *central executive* is a supervisory system that controls and regulates cognitive processes. It directs focus and targets information, making working memory and long-term memory work together. There is a broad line of psychological research which shows the impact of working memory on reading comprehension in natural languages. For example, studies show that working memory span and performance in reading comprehension measured with standardized tests is highly correlated [4, 13].

One may argue that reading code with longer identifier names requires larger cognitive capacity as more resources from working memory are claimed due to the *word-length effect*. However, participants detected semantic defects faster with long identifiers. Importantly, the long identifier names we used did not only consist of more characters but carried more information about concepts of the problem domain. This probably activated information in long-term memory [3]. This idea is supported by our finding that experienced developers benefited from longer identifier names, whereas novice programmer did not (see also 6.2). Also, if short and long identifier names carried about the same information, the positive effect of longer identifiers should disappear. In the studies by Lawrie et al. [19, 20], short identifier names were common abbreviations (e.g., `count` → `cnt`, `length` → `len`). In this case, code comprehension was comparable for abbreviated and full-word identifier names.

6.2 Impact of Programming Experience

In our study not every programmer benefited from compound identifier names equally. Only experienced programmers read and comprehended source code with longer identifier names more efficiently. The effect of programming experience on comprehension has been discussed previously.

For example, Lawrie et al. [19] hypothesized that an increased work experience leads to a better code comprehension ability in general, thus lowering the impact of the value of identifier quality. In contrast to their expectation, they did not observe any effect of work experience on code comprehension (measured with describing the functionality of the source code) but only on confidence in the description. In our study we observed that experienced programmers benefited from more informative identifier names when searching semantic defects. We will discuss three possible explanations for this effect.

First, experienced programmers are used to read code with longer identifier names. Various code conventions suggest to name functions and variables with descriptive, informative names [15, 22].

Using short (i.e., single-word) names in an experiment is rather unusual for experienced programmers and might distract them from the actual task, resulting in longer task completion times in this condition. That is, the advantage of longer identifier names may be actually a disadvantage of short identifier names.

Second, the effect of expertise might result from differences in *memory processes* (see section 6.1). Understanding source code requires an interplay between a limited-in-capacity working memory and long-term memory [8]. To circumvent the limited capacity, information needs to be stored in long-term memory or to be chunked (i.e., grouped together). Because of their experience with various problem and application domains, experienced programmers might be more efficient in mapping source code to concepts stored in long-term memory in general or to chunk related information into broader schemes (or even algorithms). More descriptive identifier names may activate those concepts faster and more effectively than short identifier names. This may leave more resources for code comprehension in working memory.

Third, novice and experienced programmers may use different *code reading strategies.* Bottom-up comprehension models propose that abstract concepts of the source code are formed by combining low-level information [26], while top-down models assume that programmers use previous exposure to the application domain to create expectations that are then mapped onto the source code [9]. Longer identifier names contain more cues to the intention behind the code, which should make it easier to apply related domain knowledge when trying to understand the code. As experienced programmers are more likely to have extensive domain knowledge that can be activated in this way, they may benefit more from informative identifiers.

6.3 Threats to Validity

Internal validity defines the extent to which a causal conclusion based on a study is warranted. We note the following limitations. First, identifiers are informative only if programmers know the problem or application domain to recall appropriate concepts from long-term memory. This has not been controlled for. Code snippets were chosen from domains that most students should be familiar with. However, there might be individual differences between participants. Second, the study was run web-based, i.e. we did not have full control over the programmers' behavior. Although we applied strong exclusion criteria (see above), it is unclear how exactly participants proceeded to solve the tasks.

External validity defines the degree to which it is warranted to generalize results to other contexts. We note the following limitations. First, the code snippets we used were not representative for large, complex software projects. For example, for each task the code snippet consisted of one class with 1 to 3 methods and a limited number of variables. That is, the scope of the source code (and thus of code comprehension) was much smaller than in a common software project, with usually a variety of classes, inheritance, and dependencies. Second, compound identifier names may not be useful for all entities in the code. Informative names may be more important for function and method names than for local variables, but this hypothesis needs to be tested in future work. Third, the effect of programming expertise was tested exploratory. Participants were divided into two groups using a median split, which might not be representative for novice and expert programmers in general. Finally, as participants knew that they were in an experimental setting, they might have behaved differently from a real situations.

6.4 Practical Implications

For experienced programmers, we observed moderate positive effects of longer but more descriptive identifier names on code comprehension. As there was neither a positive nor negative effect for novice programmers, the results of our study suggest using informative compound identifier names to improve comprehensibility of source code. Our empirical study supports approaches such as *clean code* [22] or *domain-driven design* [15]. Our findings clearly show an advantage of descriptive identifier names even when the scope of the variables is small.

This suggestion is also in line with empirical studies of code quality [10, 11], showing that code quality is associated with using identifier names composed of two to four words. Even for local variables with limited scope a descriptive name may improve code quality (but see Aman et al. [1])

7 CONCLUSION AND FUTURE WORK

The aim of the current study was to test the impact of longer but more informative identifier names on code comprehension empirically. Code comprehension was assessed by measuring the time needed to find a semantic defect in the code, assuming that such a defect can only be detected if the code has been understood. As a control condition we measured to time to find a syntax error, which does not require a full understanding of the source code.

The results show that descriptive compound identifier names improved comprehensibility reflected in shorter task completion times and fewer changes in reading direction. This was in particular the case for more experienced programmers. We propose that descriptive compound identifier names facilitate retrieving concepts from long-term memory, relieving capacity from working memory to perform the actual task.

As we varied the length of all identifiers, irrespective of their function (class, method, parameter or variable), future research may focus on investigating in which contexts using longer descriptive names improves efficiency of code comprehension. Furthermore, a better understanding of the underlying cognitive processes will help to refine guidelines and conventions for identifier naming.

ACKNOWLEDGMENTS

The authors would like to thank all participants who spend their time to support our research. This work was partially funded within the EU FP7 project *Prosperity4All* (grant agreement no. 610510) and it has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) (grant agreement no. SI 2045/2-1.).

REFERENCES

[1] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2016. Local variables with compound names and comments as signs of fault-prone Java methods. In *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality and 1st International Workshop on Technical Debt Analytics.* 4–11.

[2] Alan D. Baddeley. 2000. The episodic buffer: A new component of working memory? *Trends in Cognitive Sciences* 4, 11 (2000), 417–423.

[3] Alan D. Baddeley. 2007. *Working memory, thought, and action*. Vol. 45. OUP Oxford.

[4] Alan D. Baddeley, Robert Logie, Ian Nimmo-Smith, and Neil Brereton. 1985. Components of fluent reading. *Journal of Memory and Language* 24, 1 (1985), 119–131.

[5] Alan D. Baddeley, Neil Thomson, and Mary Buchanan. 1975. Word length and the structure of short-term memory. *Journal of Verbal Learning and Verbal Behavior* 14, 6 (1975), 575–589.

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

[7] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To camelCase or under_score. In *17th International Conference on Program Comprehension (ICPC '09)*. IEEE, 158–167.

[8] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. 2009. Identifier length and limited programmer memory. *Science of Computer Programming* 74, 7 (2009), 430–445.

[9] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.

[10] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating identifier naming flaws and code quality: An empirical study. In *16th Working Conference on Reverse Engineering (WCRE '09)*. IEEE, 31–35.

[11] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the influence of identifier names on code quality: An empirical study. In *14th European Conference on Software Maintenance and Reengineering (CSMR '10)*. IEEE, 156–165.

[12] Jacob Cohen. 1988. Statistical power analysis for the behavioral sciences . Hilsdale. (1988).

[13] Meredyth Daneman and Patricia A. Carpenter. 1980. Individual differences in working memory and reading. *Journal of Verbal Learning and Verbal Behavior* 19, 4 (1980), 450–466.

[14] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14, 3 (2006), 261–282.

[15] Eric Evans and Rafał Szpoton. 2015. *Domain-driven design*. Helion.

[16] Richard K. Fjeldstad. 1983. Application program maintenance study: Report to our respondents. *Proceedings GUIDE 48, 1983* (1983).

[17] Johannes C. N. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2017. Shorter identifier names take longer to comprehend. In *24th International Conference on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE, 217–227.

[18] Anthony R. Jansen, Alan F. Blackwell, and Kim Marriott. 2003. A tool for tracking visual attention: The Restricted Focus Viewer. *Behavior Research Methods* 35, 1 (2003), 57–69.

[19] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a name? A study of identifiers. In *14th International Conference on Program Comprehension (ICPC '06)*. IEEE, 3–12.

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

[21] Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*.

[22] Robert C. Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[23] Phillip Anthony Relf. 2004. Achieving software quality through source code readability. *Quality Contract Manufacturing LLC* (2004).

[24] Teresa M. Shaft and Iris Vessey. 1995. The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research* 6, 3 (1995), 286–299.

[25] Bonita Sharif and Jonathan I. Maletic. 2010. An eye-tracking study on camelCase and under_score identifier styles. In *18th International Conference OnProgram Comprehension (ICPC '10)*. IEEE, 196–205.

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