Empir Software Eng (2019) 24:417–443
https://doi.org/10.1007/s10664-018-9621-x

Shorter identifier names take longer to comprehend

Johannes C. Hofmeister[1] · **Janet Siegmund**[1] · **Daniel V. Holt**[2]

Published online: 26 April 2018

Abstract Developers spend the majority of their time reading code, a process in which identifier names play a key role. Although many identifier naming styles exist, they often lack an empirical basis and it is not clear whether short or long identifier names facilitate comprehension. In this paper, we investigate the effect of different identifier naming styles (single letters, abbreviations, and words) on program comprehension. We conducted an experimental study with 72 professional C# developers who had to locate defects in source code snippets. We used a within-subjects design, such that each developer worked with all three versions of identifier naming styles, and we measured the time it took them to find a defect. We found that word identifiers led to a 19% increase in speed to find defects compared to meaningless single letters and abbreviations, but we did not find a difference between letters and abbreviations. The results of our study suggest that code is more difficult to comprehend when it contains only letters and abbreviations as identifier names. Words as identifier names facilitate program comprehension and may help to save costs and improve software quality.

Communicated by: Andrian Marcus and Gabriele Bavota

This article extends a previous conference paper presented at the 24th International Conference on Software Analysis, Evolution and Reengineering (Hofmeister et al. 2017). See the end of Section 1 for details.

✉ Johannes C. Hofmeister
johannes.hofmeister@uni-passau.de

Janet Siegmund
siegmunj@fim.uni-passau.de

Daniel V. Holt
daniel.holt@psychologie.uni-heidelberg.de

[1] University of Passau, Innstrasse 33, 94032 Passau, Germany

[2] Heidelberg University, Hauptstrasse 47-51, 69117 Heidelberg, Germany

Keywords Identifier names · Program comprehension · Professional C# developers · Psychology · Defect detection · Software quality

1 Introduction

Identifier names are important for program comprehension. Their relevance has been discussed for more than 30 years now, for example, by Brooks (1983), and Soloway and Ehrlich (1984), who explained that they serve as *key beacons* to *program plans*, which activate higher level knowledge about the program and facilitate program comprehension.

Identifiers make up large parts of source code. For example, Deissenboeck and Pizka found that 33% of all tokens of the code of Eclipse 3.1.1 were identifiers which accounted for 72% of all characters in the code (Deissenboeck and Pizka 2006). In principle, developers are free to choose identifier names at their own discretion, which can lead to varying results depending on experience, skill, and mood (Sneed 1996). In most modern programming languages, identifier names are limited only by few syntactic constraints (e.g., they are restricted to alphanumeric characters), and the concrete word can be chosen almost arbitrarily (i.e., as long as the used string is not a reserved keyword). As a result, developers might be inclined to use single letters as identifier names, for example, to save typing effort.

When the chosen identifier names are meaningless, developers are most likely slower in comprehending the program's functionality, especially when they are unfamiliar with the code (Soloway and Ehrlich 1984). To alleviate this well-known problem, communities of many programming languages promote style guides, and companies establish specific naming conventions, all with the goal to improve understandability and maintainability of source code.

Unfortunately, style guides and conventions address only superficial features of identifier names. For example, most style guides prominently encourage the use of a particular separation style (i.e., camelCase or under_score) for compound identifier names. However, semantic properties are often left unmentioned or are limited to the type of word (e.g., classes should have nouns as names; MSDN 2016). Furthermore, conventions often lack a sound empirical basis (Tichy 1998). Thus, it is unclear how important the influence of identifier naming on comprehensibility and maintainability really is. Understanding the different aspects of identifier naming can help to choose a particular naming style that supports program comprehension, which in turn can help to improve productivity, enhance software quality, and reduce cost.

In practice, programmers do not just use random strings as identifier names but they usually name identifiers after a thing or purpose. An identifier is not only a syntactic element but also a symbol that associates a concept from the problem domain with an entity in the code. For example, naming a class `Person` will associate the concept of a person with its class-representation in the code.

Ideally, an identifier name designates a concept from the problem domain, but it does not have to. For example, when a class is named `DataInfoContainer`, but it represents a customer of a shopping site, it becomes difficult to deduce that it carries a person's information, because the name is unrelated to the concept (i.e., that it contains data of a customer). This problem has been addressed by other authors. For example, Deissenboeck and Pizka classified bad identifier names along the dimensions of *correctness* and *consistency* (Deissenboeck and Pizka 2006). From this viewpoint, an identifier named `DataInfoContainer` is *incorrect*, because its name is neither a subordinate nor superordinate name of the designated concept *person*. Deissenboeck and Pizka's work provides an

excellent meta-linguistic framework to discuss the issue, but does not evaluate empirically how inconsistent or incorrect identifier names affect developers.

In this paper, we describe an experimental study in which we quantified the impact of *length* and *semantics* of identifier names on program comprehension. To this end, we invited professional software developers to find defects in code that followed different identifier naming conventions (single letters, abbreviations, and words).

We make two contributions in this paper:

- Empirical evidence that full words as identifier names positively influence comprehensibility and maintainability of source code
- A replication package of our experiment and data to help other researchers validate and extend our results.[1]

This article extends a previous conference paper that was presented at the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER; Hofmeister et al. 2017). For the present publication we added an analysis of the participants' movement of visual focus in Section 5.3 along with a corresponding discussion. The data used for the additional analysis had been obtained during the original experiment but the analysis was omitted for brevity in Hofmeister et al. (2017).

In the following section, we explain why both single letters and words can in principle have a positive effect on program comprehension. In Section 3, we derive and explain our research hypotheses. Section 4 describes the experimental setup. We report our findings in Section 5 and discuss our results in Section 6. We address threats to validity in Section 7.

2 Word-length and semantics

In this paper, we focus on two main aspects of identifier names, *length* and *semantics*. An identifier name's *length* refers to the number of letters in the word used. We use the term *semantics* to refer to the *meaning* of words in the linguistic sense, rather than a token's semantics in the context of a programming languages, for example, the behavior of the ++ operator.

On the one hand, developers might optimize their code for brevity and choose short identifier names, because they want to reduce typing effort or because an algorithm is implemented close to a mathematical equation. Single letters are the shortest possible identifier names and abbreviations are also common, for example, `configuration` can be shortened to `config`, or `cfg`. On the other hand, identifiers should convey the concept they represent as clearly as possible (Anquetil and Lethbridge 1998), which is best achieved by using words that represent the concept (e.g., a customer is best represented by an identifier named `customer`, not `data`). Words are longer than abbreviations, but their meaning is clearer. Optimizing for a very short or very meaningful identifier naming style should not be left to chance or personal preference, but is ideally based on empirical results with a focus on human developers.

Psychological research has long been studying the readability and comprehensibility of natural-language texts, and we can find results supporting the use of both short strings and full words as identifier names. On the one hand, very short identifier names, such as abbreviations and letters, require less cognitive capacity as predicted by the *word-length effect*

[1] http://brains-on-code.org/

(Baddeley et al. 1975). This effect describes that lists of short strings are easier to remember than lists of long strings. Thus, developers' performance could be positively affected by very short identifier names because more items fit into working memory, which helps developers to keep a better overview of the code.

On the other hand, further findings indicate that length affects the cognitive processing of text. Longer strings take longer to pronounce (Balota and Chumbley 1985) and have a higher naming latency (i.e., are uttered with delay) than shorter strings (Weekes 1997). This affects non-words, such as arbitrary strings or nonexistent words (e.g., "awek", "enemenemoo"), as well as low-frequency words (e.g., "penultimate", or "hypochondriasis"), but not common words (e.g., "awake", "hat"), which is called the *word-frequency effect*. Studies that controlled for the ease of articulation showed that not the process of articulation, but rather the required synthesis of the string's phonetics is responsible for the delay (Weekes 1997).

The Dual Route Cascade Model (DRC; Coltheart et al. 2001) explains these findings: words that are common in natural language (e.g., "awake", "hat") are stored in a *mental lexicon* (i.e., a dictionary that maps concept to read or spoken words). When they are perceived they can be accessed via the *lexical route* (i.e., they activate a concept or meaning from the mental lexicon). Words that do not exist in this mental lexicon cannot be immediately accessed, because their phonetics have to be synthesized on the fly, activating the *phonetic-graphemic route*, a process that is serial in its nature and therefore depends on word length.

When comprehending source code, working memory plays a crucial role, because developers have to work with several programming constructs (e.g., variables and methods) in parallel. Human working-memory is limited,[2] and a word's semantics can help to relieve cognitive resources through a process called *chunking* (Baddeley et al. 1975) in which items are regrouped to more meaningful units.

Additionally, a word's semantics influence how adjacent words are processed, an effect called *semantic priming* (Collins and Loftus 1975). This effect may play a role in discovering inconsistencies. When code is too abstract, it might become more difficult to detect semantic defects, as illustrated in Listings 1 and 2:

Listing 1 A login function using letter identifier names

```
def v(u, p):
    u1 = d.u()
    p1 = d.p()
    return u == u1 and p == u1
```

Listing 2 The same function using word identifier names

```
def login(username, password):
    user = db.username()
    pass = db.password()
    return username == user and password == user
```

The codes are equivalent in their structure; only the identifier names have changed. Both codes are syntactically valid, but the inconsistency is (arguably) easier to detect in Listing 2, when more concrete contextual semantics are present. Since Listing 1 uses only abstract

[2]Miller (1994) originally argued for a capacity limit of about 7 ± 2 items, while newer research shows that core working memory capacity is more likely limited to 3 to 5 items (Cowan 2001).

identifier names, it is difficult to detect the semantic defect. However, altering the code snippet as in Listing 1 reveals that the wrong comparison is made (`password` against `user`). Thus, meaningful, full word identifier names activate context semantics, which allow developers to evaluate code against its purpose.

To summarize, brevity and semantics of identifier names seem to contradict each other regarding their effect on program comprehension, and it is unclear from which effect developers could benefit more.

3 Hypotheses

In this study, we address the following research question:

How do identifier name length and semantics affect developers' performance during program comprehension?

If program comprehension benefits most from an identifier name's *semantics*, then comprehension of code using words as identifier names, such as `customer` or `request`, should be faster than code using abbreviations (e.g., `cst` or `rqs`) or unrelated single letters (e.g., `a` or `b`). If an identifier name's *length* is in fact more important for comprehension than its semantics, then shorter but less meaningful identifier names, (e.g., `a` or `rqs` instead of `request`) should make code faster to understand than code using words.

This question has been investigated by a related study by Lawrie et al. (2006), who, focusing on the effect of identifier length, suggest that multi-word identifiers and abbreviations are easier to comprehend than identifiers truncated to a single letter. Lawrie et al. (2006) measured program comprehension by asking participants to describe the purpose of presented functions, which where later rated on a scale from 0 to 5, with 5 indicating a correct understanding. The authors found a statistically significant effect of identifier style for 3 out of 12 code snippets. They do not report a main effect of identifier style, but focus on the interaction between code type and identifier naming style in their analysis. The present work extends the experiment by Lawrie et al. (2006) using a different task (the speed to find a bug) and introducing several methodological improvements described below.

Determining the impact of length is difficult, because different predictions can be drawn from the aforementioned psychological effects. On the one hand, the processing of non-words depends on their length. Thus, abbreviations, compared to single letters should require an increased effort to process, as they are longer than single letters, which should decrease program comprehension performance. On the other hand, the residual semantic properties of abbreviations might facilitate higher cognitive processing (i.e., ease of lexical access, semantic priming) and therefore, abbreviations should lead to faster comprehension than code with unrelated single letters as identifier names. Still, it can be expected that the meaning of words is accessed lexically, so abbreviations should still be slower to comprehend than full words.

We reasoned that identifier names' semantic properties affect in-depth understanding of code, rather than its shallow perceptual processing. Therefore, the performance of working with code without trying to comprehend it should not be positively influenced by the presence of full words. However, it is possible that, on such a low level of processing, the reduced amount of code leads to faster processing. When the meaning of an identifier is irrelevant (e.g., to find a missing semicolon), then fewer characters imply less code to read and thus could even improve the performance in such tasks.

In addition to the effect of identifier semantics on the fluency of cognitive processing, we also expect an effect of identifier naming style on comprehension performance via simple

forgetting. In contrast to words, unrelated single letters do not contain any semantic cue as to what they designate (e.g., whether `a` represents a customer, a call center agent, a counter, or yet something else). If during the course of reading a piece of source code a developer forgets what a variable stands for, a single letter identifier name is of little help. Instead, the meaning of the identifier needs to be reconstructed from the context, by active remembering, or by reading explanatory comments, all of which take additional time. We expect a similar effect for abbreviated identifier names, which also require mentally reconstructing or remembering the abbreviated word. However, due to the cue function of the abbreviation, the difference to full words may be less pronounced.

To answer our research question, we designed a controlled experiment, in which we evaluated the following hypotheses:

$RH_{Semantic}$:	Words as identifier names lead to faster comprehension of source code than abbreviations and unrelated single letters.
RH_{Syntax}:	Identifier naming style has no effect on locating syntactical errors

3.1 Independent variables

We tested comprehension performance for three conditions that represent different points on the spectrum of length versus semantics as described above. Long and meaningful identifier names, such as full words, lie at one end of the spectrum, meaningless single letters at the other, while abbreviations fall in between. For brevity, we henceforth label these conditions *letters*, *abbreviations*, and *words*.

- *Letters* are the shortest possible identifier names. We chose letters that were unrelated to the identifier's purpose in the code to eliminate any semantic content.
- *Words* carry more semantics but also have an increased length. We chose single word identifier names that can be found in a dictionary and designate a single concept in the program.
- *Abbreviations* form a compromise between these two. They were constructed from the original word identifier names and are shorter but still contain semantic traces.

3.2 Dependent variables

We operationalized the *performance of comprehension* by measuring how long developers investigated a snippet of code until they had found a defect. We assumed that a semantic defect in code can be corrected only when it is understood, because developers cannot evaluate the consequences of their changes otherwise. We required developers to indicate when they had found the defect to approximate the exact moment of comprehension.

As a control condition, we tested whether identifier naming style affects the performance in tasks in which no deep understanding of the code is required. This allowed us to evaluate whether our conditions interfered with program comprehension, or whether some other process is being measured. 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 deep understanding of its identifier names' meanings to be corrected.

In order to explore the process of reading source code, we furthermore employed a *restricted focus viewer* (Jansen et al. 2003), which allowed us to record how much time developers spent looking at different parts of the code. In line with the main research hypotheses, we expected that non-word identifiers would slow down the reading process

Table 1 Experiment overview

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

even in parts of the code that do not contain a defect. Related, we expected that particularly the single-letter identifiers would lead to more frequent backward movements to the comments section to refresh developers' memory of what the letter identifier stands for. Again, these effects should only be present when trying to find semantic defects, not for syntactic defects.

4 Study design

We tested our hypotheses in a web-based experimental study. Participants were asked to find and correct a defect in six snippets of code. We measured how much time they spent on the task. The scope of our experiment was to *investigate identifier naming styles* for the purpose of *quantifying their effect on program comprehension* with respect to *participants' comprehension speed* in the context of *C# development*. An overview of the experiment following the template provided by can be found in Table 1.

4.1 Participants

The sample consisted of 72 professional C# developers who were between 20 to 51 years old, with a median of 35 years. Their overall programming experience ranged from 4 to 35 years, with a median of 14 years. Their median experience with C# was 8 years, ranging from 2 to 15 years. We invited them to our experiment via online platforms, such as Twitter and Xing. Additionally, we approached participants of German technology industry conferences.

Overall, 221 people started to participate in the experiment and 135 completed it. The records of 63 participants were removed after applying exclusion criteria to ensure high data quality (see below). The remaining participants were randomly assigned to the experimental sequences. The participants' details are displayed in Table 2.

To obtain the data, we implemented a web application, which is available at the project's website. Since we conducted the experiment online we could not guarantee an undisturbed working environment. To reduce this threat to validity we removed participants who

Table 2 Descriptive participant data

	Category	Percent
School Education	12 to 13 years	81%
	10 years	11%
	Other	8%
Higher Education	Master's	42%
	Bachelor's	18%
	Vocational training	18%
	No higher ed.	15%
	Other	7%
Employment Status	Employed	81%
	Freelance	17%
	Student	2%
Job Description	Software Developer	51%
	Consultant	12.5%
	Software Engineer	12.5%
	Project Manager	10%
	Software Architect	8%
	Other	6%

reported that they had encountered distractions or mentioned distracting circumstances in a comment field (e.g., "my boss came in") on the last page of the study.

To reduce further threats to validity, we controlled for other factors. For example, participants were required to have sufficient natural language skills to understand instructions and code comments and had to have more than one year experience with C# in practical use. They provided self-ratings of their language proficiency on a scale from 1 to 6 for German and English. Since the code was written using English identifier names and the instructions were given in German, the data of participants with ratings below 4 in either category were excluded. We targeted professional developers and avoided selecting students; however, in the final sample the records of two students remained. They stated appropriate experience with C# to be considered professional developers.

Finally, on the last page of the study we asked participants whether they had worked on the experimental tasks conscientiously, or whether they were just curious and wanted to take a look. All exclusion criteria and number of affected records are listed in Table 3. When one criterion applied to a participant, their whole dataset was removed and their trial data was not used. For our results and analyses, we used only participants to whom these criteria did not apply, and who provided a complete set of six trials.

Table 3 Exclusion criteria

		Criterion	n
Language	German (1 - 6)	< 4	2
Level	English (1 - 6)	< 4	9
Programming	C# Skill (1 - 5)	< 4	24
Experience	C# Experience (years)	< 1	8
Behavior	Encountered distractions?	Yes	17
	Worked on task conscientiously?	No	1
	Attempts to succeed (per trial)	> 3x	15
	Freeze, AFK (no interaction)	> 1 min	4
	Too Slow (time per trial)	> 10 min	14
Other	Participated in pilot study?	Yes	4
	Participated before?	Yes	8
	Total (criteria not mutually exclusive)		63

4.2 Task

The participants' task was to find one defect in a snippet of code. The task was repeated six times: after they had worked on three snippets finding *semantic defects*, participants were asked to work on three more snippets but to look for *syntax errors* now.

To gather coarse-grained data about participants' visual focus, we used an implementation of the restricted focus viewer (Jansen et al. 2003), which also helped us to detect distracted participants. Participants' view on the code was limited to 7 lines at a time (approximately one third of the complete snippet), but the frame could be shifted up and down using the arrow keys to reveal different parts of the code (Fig. 1). We called this feature the *letterbox*, because it mimics spying through the letter slit in a door or mailbox.

When participants had found the defect, they pressed the space bar. This keypress froze the letterbox, and opened a dialog screen, in which the participants entered the line number of the defect, a description, and a correction (Fig. 2). We measured how long participants looked at the code until they indicated that they had found the defect. We subtracted the time spent answering the dialog and only evaluated the time that participants interacted with code. This way, we analyzed only the time required to comprehend the code. Participants who had failed to find the defect in a snippet after three attempts were allowed to finish the experiment, but their data were excluded.

We used the time required to find semantic defects as a measure of program comprehension. The response is easy to score for correctness and it has a well-defined time point, allowing reaction time analyses. Furthermore, 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. Because the study was conducted online, we ruled out think-aloud protocols. Locating defects is a common programming task, which renders it a relevant target for studying program comprehension.

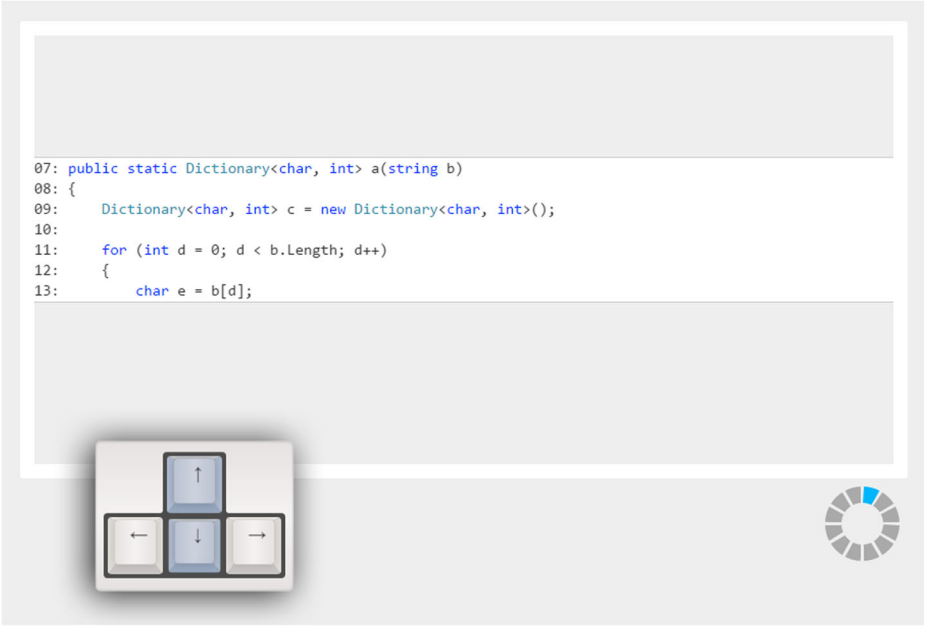


Fig. 1 The *letterbox* limited participants' view to seven lines of code at once. They were able to shift the view using the arrow keys

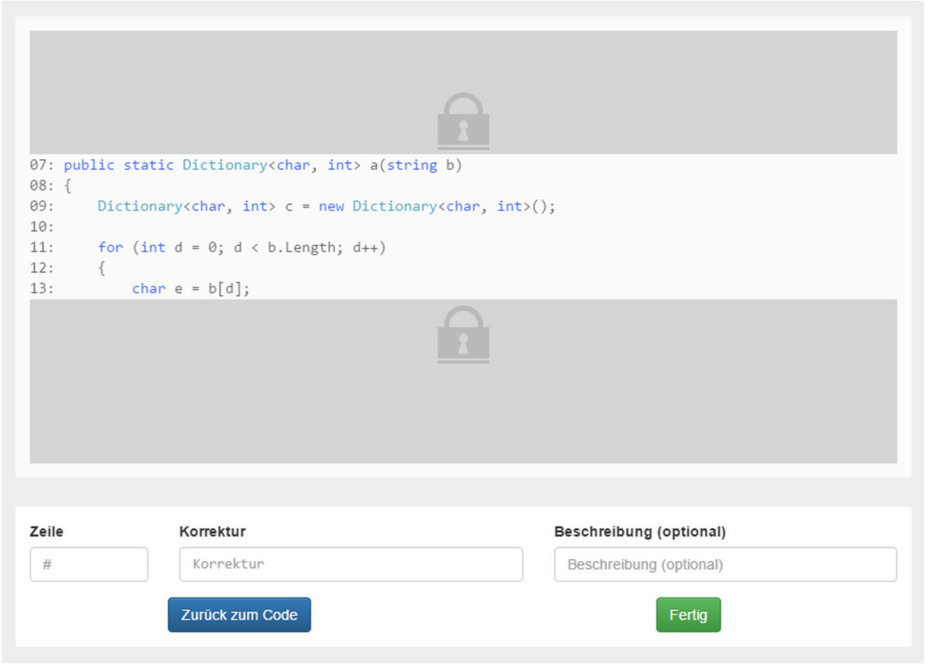


Fig. 2 Participants were instructed to press space to indicate that they had found the defect. This fixated the letterbox and opened a dialog

```
// ConcatLists: Concatenates two lists of the same length
// start: collection of elements at the start
// end: collection of elements to append
// length: length
// result: result
// index: index
// first: first
// second: second

public static int[] ConcatLists(int[] start, int[] end)
{
    int length = start.Length;
    var result = new int[length * 2];

    for (int index = 0; index < length; index++)
    {
        int first = start[index];
        int second = end[index];

        result[index] = first;
        result[index + 1] = second;
    }
    return result;
}
```

Listing 3 Snippet using words as identifier names

4.3 Materials

We initially created eleven new code snippets containing simple algorithms to ensure that no participant had seen the materials before. The snippets needed to be simple enough to be comprehensible in a reasonable time frame, but complex enough for defects to "hide" in the code. Each snippet consisted of a self-contained static function with a length of 15 lines. Listing 3 shows an example. We limited the code to language features from C# 2.0, such as loops, conditionals, and basic .NET API calls. We avoided more complex structures, such as recursion, or specific APIs (e.g., Language-Integrated Query; LINQ), to avoid bias due to extensive C# experience. Each snippet came in three versions, in which the identifier names were changed to either words, abbreviations or meaningless single letters. Examples are shown in Table 4. Each snippet had one version with a *semantic defect* and a one with a *syntax error*. The errors were placed in similar locations in the code to avoid bias due to different locations of the errors.

Each snippet was built with expressive word identifier names first. From this version, two derived versions were generated by replacing the identifier names in an automated process. Abbreviations were generated by keeping the first character of the original word identifier name. Then the automated process removed all vowels and left the first two remaining consonants in place (e.g., `request` became `rqs`). In a few cases, this caused the identifier names to collide, which we resolved by replacing some identifiers with fixed alternatives.

Table 4 Examples of the different identifier naming styles used

Style	Example	Design
Word	`request`, `histogram`	Initial code
Abbreviation	`rqs`, `hst`	Derived from word, first letter and consonants, avoided collisions
Letter	`a`, `b`	Chosen alphabetically

Listing 4 Snippet using abbreviations as identifier names

```
// Cnc: Concatenates two lists of the same length
// str: collection of elements at the start
// end: collection of elements to append
// len: length
// rsl: result
// idx: index
// frs: first
// scn: second

public static int[] Cnc(int[] str, int[] end)
{
    int len = str.Length;
    var rsl = new int[len * 2];

    for (int idx = 0; idx < len; idx++)
    {
        int frs = str[idx];
        int scn = end[idx];

        rsl[idx] = frs;
        rsl[idx + 1] = scn;
    }
    return rsl;
}
```

Additionally, we chose to replace common identifier names with their conventional abbreviations, such as `len` for `length`, and `min` for `minimum`, because we did not want to violate the participants' expectations too much and maintain plausibility.

In the letter version, identifiers were named alphabetically, in the order of occurrence, while ensuring the validity of the code. We decided on alphabetical replacement to guarantee that the identifiers did not resemble the original identifier names in any way. The standard .NET API was left intact (e.g., identifiers such as `List` were not abbreviated). Each function was commented on top. The first line contained an explanatory description of the method's desired functionality. The following lines documented the variables and, in the abbreviation and letter versions, showed their original meaning.

We evaluated the snippets' suitability in a pilot study. Participants were shown three snippets with word identifier names and we measured the time until participants found a

Listing 5 Snippet using letters as identifier names

```
// a: Concatenates two lists of the same length
// b: collection of elements at the start
// c: collection of elements to append
// d: length
// e: result
// f: index
// g: first
// h: second

public static int[] a(int[] b, int[] c)
{
    int d = b.Length;
    var e = new int[d * 2];

    for (int f = 0; f < d; f++)
    {
        int g = b[f];
        int h = c[f];

        e[f] = g;
        e[f + 1] = h;
    }
    return e;
}
```

semantic defect. The data of the pilot study were not used to answer our research question. We had recruited two different samples of participants online, but could not prevent a slight overlap between the samples. Records of people who took part in the pilot study were excluded. We removed five snippets after the pilot study, because the measured times exposed too much variance (i.e., the difference between fast and slow participants was too high), or because they where too difficult (i.e., all participants were comparatively slow).

Listings 3, 4, and 5 show three snippet versions, all of which show the same algorithm that naïvely concatenates two lists. The defect resides in Line 21 and its correction could be: `result[index + length] = second;`.

4.4 Procedure

Participants were invited to a public website where they found an introduction text, legal information related to informed consent, and a privacy statement. From there, they entered the experiment, starting with questions about their education, employment status, and professional experience.

They continued with a tutorial that gradually familiarized them with the experiment. After the tutorial, an overview of the upcoming task was presented. On the next screen, participants were instructed to press the space bar to start the trial. The participants inspected the code, searching for a defect. When they had found the defect, they pressed the space bar again, opening the aforementioned correction dialog. After filling out the dialog, participants received feedback whether their answer was correct to motivate them to continue.

After the experiment, we asked for some demographic data, and whether or not the participants had been distracted during the experiment.

4.5 Design

The goal of our experiment was to quantify the effect of identifier naming styles on program comprehension. To isolate the effect as much as possible, we controlled several factors that could affect comprehension performance, namely the effects of inter-individual differences

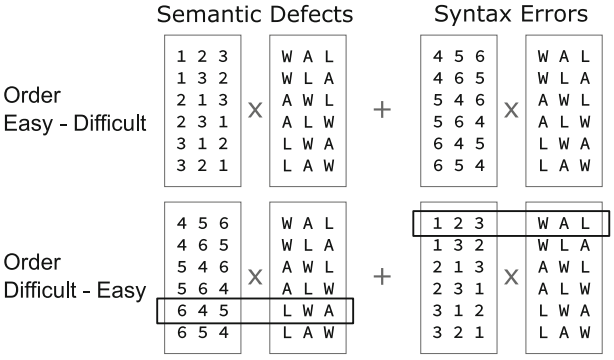


Fig. 3 We used a balanced design to control for learning effects and effects caused by our stimulus material. Each block indicates a group of snippets. Group Easy contains snippets 1,2, and 3, Group Difficult contains 4,5, and 6. Snippets within each groups were permuted and multiplied with the permuted identifier style: word (W), Abbreviation (A), Letter (L), resulting in 36 combinations per task. Each participant received a random combination for each task (i.e., 6 trials in total) for example: 6L-4W-5A-1W-2A-3L

between participants, the difficulty of the snippets, and order effects. This resulted in the design illustrated in Fig. 3.

4.5.1 Inter-individual differences

To control for inter-individual differences, we used a within subjects-design, such that every participant saw all realizations of the different identifier naming styles. This compensated participants' different skill levels, for example, that slow readers would be slower in every task.

4.5.2 Material effects

To reduce side-effects caused by our materials (e.g., a complex problem takes longer to comprehend, but not because of its identifier naming style), we grouped the snippets into two sets of three snippets each, depending on their difficulty (*Group Easy* and *Group Difficult*) as established in the pilot study. In the final experiment, half of the participants were shown three snippets with semantic defects from Group Easy first, followed by three snippets containing syntax errors from Group Difficult, vice versa for the other half of participants. Furthermore, we permuted the order of snippets within each group to counterbalance snippet-specific difficulty and order effects.

4.5.3 Effects of condition order

To reduce the effects of condition order, which may lead to increased or decreased performance over the course of the experiment, we also permuted the order of identifier naming styles in each group. Table 5 shows an example trial for one participant, with Group Easy consisting of Snippets 1, 2, and 3, Group Difficult consisting of Snippets 4, 5, and 6.

Combining and permuting these factors, we generated 72 different sequences of snippets (3! identifier naming style $\times$ 3! snippet order $\times$ 2 difficulty order), which also defined the sample size.

In summary, every participant saw:

- Three semantic defects first, then three syntax errors
- All identifier naming styles
- All six snippets, encountering each snippet only once

During the pilot study, we had observed that syntax errors where found much faster than semantic defects. Thus, to prevent participants from being discouraged by the upcoming amount of work and drop out of the experiment, we explained that the last three items

Table 5 Example trial sequence for one participant

Sequence	Group	Task	Snippet	Style
1	Difficult	Semantic Defect	6	Letter
2	Difficult	Semantic Defect	4	Word
3	Difficult	Semantic Defect	5	Abbreviation
4	Easy	Syntax Error	1	Word
5	Easy	Syntax Error	2	Abbreviation
6	Easy	Syntax Error	3	Letter

Table 6 Duration of interaction with code by identifier naming style in `m:ss` (minutes, seconds)

	Semantic		Syntactic	
Style	*Median*	*IQR*	*Median*	*IQR*
Word	1:24.48	1:12.78	0:39.42	0:49.00
Abbreviation	1:38.57	1:05.37	0:36.71	0:53.92
Letter	1:40.36	1:24.87	0:35.74	0:30.22

together (syntax errors) required about as much time as one of the previous items (semantic defects). We explained that the total duration of the experiment was 20 to 30 minutes altogether.

5 Results

To test our hypotheses, we analyzed the response time data of participants (i.e., the time they viewed code until they pressed the space bar). In this section, we first present data preparation and the descriptive statistics, and then the test of our hypotheses. For convenient interpretation, tables and figures in this section show group statistics for each condition across all participants. Our statistical tests rely on within-subjects comparisons to take into account speed differences between participants.

5.1 Data preparation and descriptive statistics

Table 6 shows a summary of the raw reaction time data split by identifier naming styles. The distribution of data is skewed, such that fast responses accumulate on the left and with a tail of slow responses on the right side of the distribution, a phenomenon common for reaction times (Ratcliff 1993). Under these circumstances common descriptive statistics, such as mean and standard deviation, become difficult to interpret. In this case, the median as a measure of central tendency and the interquartile range (IQR[3]) as a measure of dispersion are more suitable (Whelan 2008).

The presence of outliers can reduce the power of experimental analyses. According to (Ratcliff 1993, p. 510), outliers are "response times generated by processes that are not the ones being studied"; for example, participants could have been distracted, or might have lost attention. There are several ways to reduce the impact of outliers and retain the power of statistical tools, including trimming, winsorizing, and transforming. Trimming removes outlier data points above a certain cutoff threshold, winsorizing replaces them with the threshold value, and transforming the data changes the data distribution (Ratcliff 1993; Leonhart 2009).

We chose to transform the data using an inverse transformation, which represents a good compromise between reducing the impact of outliers, data retention, and interpretability (Ratcliff 1993). The transformed values express *defects per minute* rather than *minutes per defect*, that is, the speed of finding defects. The data are displayed in Table 7. To ensure that the normality requirement for parametric testing was sufficiently fulfilled, we monitored

[3]The IQR is defined as $Q3 - Q1$, where the slowest 25% of response times lie below Q1 (first quartile) and the fastest 25% above Q3 (third quartile)

Table 7 Response speed (defects per minute) during the semantic and syntax tasks by identifier naming style

	Semantic		Syntactic	
Style	*M*	*SD*	*M*	*SD*
Word	0.78	0.42	1.76	1.13
Abbreviation	0.65	0.31	1.81	1.31
Letter	0.66	0.39	1.96	1.39

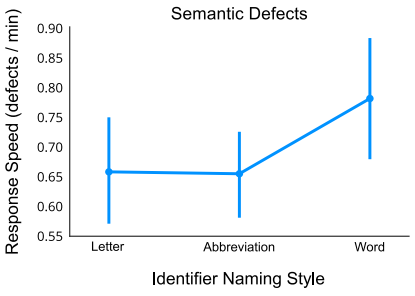
the absolute skewness value for all variables used in the statistical tests, which was less than one in all cases.

5.2 Hypothesis testing

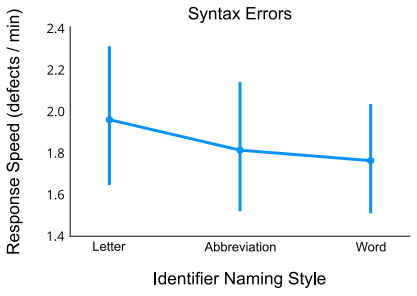
We calculated inferential statistics for *semantic defects* and *syntax errors* separately. Our analysis focuses on the semantic task because the syntax task was mainly designed as control condition. The critical significance level was set at $\alpha = .05$ for all tests. As standardized effect sizes we report Cohen's d_z with a correction for correlated observations for paired comparisons and generalized eta squared (η_g^2) for analyses of variance (Cohen 1988; Bakeman 2005).

5.2.1 Semantic defects

We tested the overall effect of identifier naming style on the speed of finding semantic defects using an analysis of variance (ANOVA) with one within-subjects factor. As expected, we observed an overall effect of identifier naming style, $F(2, 142) = 4.46$, $p = .01$, $\eta_g^2 = .02$. To locate the effect precisely, we followed up on this analysis with paired comparisons. Semantic defects were found faster when using word identifiers compared to both single letters, $t(71) = 2.47$, $p = .02$, $d_z = 0.31$ and abbreviations, $t(71) = 2.47$, $p =$



(a) Participants detected semantic defects faster when code contained words as identifier names in comparison to single letters or abbreviations.



(b) Although there appears to be a small effect of identifier names on finding syntax errors it is not statistically significant.

Fig. 4 Effect of identifier naming style on response speed. Vertical bars show 95% confidence intervals

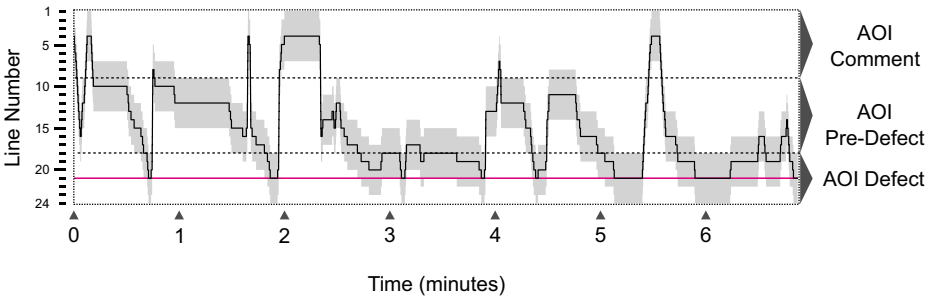


Fig. 5 Plot of a participant's letterbox movements over time. The gray area in the plot indicates the seven lines of code visible through the letterbox. The black line indicates the center of view. This particular participant spent 6 minutes and 55 seconds trying to find a semantic defect in a snippet with abbreviated identifiers (see Listing 4). The participant frequently visits AOI_{Defect} and occasionally revisits AOI_{Comment}

.02, $d_z = 0.34$, see Fig. 4a.[4] There was no difference between abbreviations and single letters, $t(71) = 0.07$, $p = .94$, $d_z = 0.01$.

The difference between words and abbreviations or letters represent a small to medium-sized effect according to Cohen (1988). Expressed as a relative speed advantage, the median within-subjects improvement was 21% for word identifiers compared to single letter identifiers and 13% for word identifiers compared to abbreviations. When combining response times for single letters and abbreviations, the relative speed advantage of word identifiers was 19%.

5.2.2 Syntax errors

We did not find a significant effect of identifier naming styles on the detection of syntax errors, as illustrated in Fig. 4b, $F(2, 142) = 0.80$, $p = .45$, $\eta_g^2 < .01$. Considering that our study has 80% statistical power to detect effect sizes as small as $\eta^2 = .03$, we interpret this result as support for the assumption that identifier names have at most a negligible effect on finding syntax errors.

5.3 Visual focus

As a proxy for the focus of participants' visual attention during the task, we analyzed the movements of the letterbox, which showed only seven lines of code at once while hiding all other lines (see Fig. 1).

The letterbox had to be moved one line at a time using the cursor keys. To illustrate, Fig. 5 shows the letterbox movements of one participant over time. Participants could not scroll the letterbox past the boundaries of the code displayed. Upon starting each task, the first seven lines of the snippets were visible in the letterbox. We analyzed the letterbox position in analogy to eye tracking data.

The code was split into three main *areas of interest* (AOI): AOI_{Comment}, $AOI_{\text{Pre-Defect}}$, and AOI_{Defect}. AOI_{Comment} comprised all comment lines at the top of the snippet,

[4]The t-values of these two tests are by chance identical when rounded to two decimal places. The standardized effect sizes differ due to the correction for correlated observations.

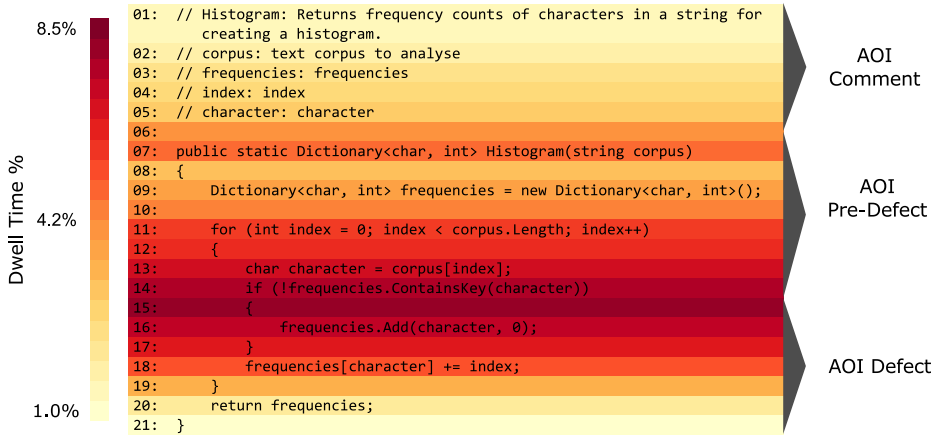


Fig. 6 Exemplary heat map of relative dwell times for each line in a particular snippet. Overall, participants fixated the lines before the defect more often than the comments or the bottom of the snippet

$AOI_{\text{Pre-Defect}}$ comprised the lines of code before the defect becomes visible in the letterbox, and AOI_{Defect} comprised the lines where the defect was visible in the letterbox. Each letterbox position was attributed to the AOI that contained the center line of the letterbox. As the lines below AOI_{Defect} were few and received little consideration (1.27% of the total time spent on the task), we removed them from the analysis, as well as the blank line separating comments and actual code. Figure 6 illustrates the AOIs with a heat map showing the average time spent on each line for this particular snippet.

For our analyses, we evaluated the AOIs' *dwell times*, *first-pass reading times* (FPRT), and *AOI 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. FPRT represents the time participants took to navigate the letterbox from entering the code section (after reading the comments) until the defect first appeared in the letterbox.

Table 8 Total dwell times by AOI and identifier naming style in seconds

		Semantic		Syntactic	
AOI	Style	*Median*	*IQR*	*Median*	*IQR*
Comment	Word	25.89	19.68	3.56	5.89
	Abbreviation	27.61	18.36	4.80	8.20
	Letter	28.12	26.70	4.22	6.86
Pre-Defect	Word	30.07	29.68	21.45	27.05
	Abbreviation	32.13	27.84	21.02	23.11
	Letter	37.51	40.04	17.81	15.61
Defect	Word	37.09	35.41	16.33	15.74
	Abbreviation	34.94	31.60	13.57	16.81
	Letter	28.98	45.89	12.35	17.02

As dwell times, FPRT, and the frequency of visits to AOIs were strongly right-skewed, we applied a logarithmic transformation for inferential tests. For ease of interpretation, descriptive statistics report untransformed values.

5.3.1 Dwell time

For a broad overview of the allocation of visual attention, we compared the total dwell time (i.e., the total time spent in each AOI) for the different identifier styles and the two different tasks using a full factorial repeated-measures ANOVA. The data are displayed in Table 8. Dwell times considerably differed depending on AOI, $F_{\text{AOI}}(2, 142) = 93.43$, $p < .001$, $\eta_g^2 = .13$ and task type, $F_{\text{Task}}(1, 71) = 273.57$, $p < .001$, $\eta_g^2 < .26$. However, there was no statistically significant interaction of identifier naming style and AOI in either task, $F_{\text{Sem}}(4, 284) = 1.34$, $p = .26$, $\eta_g^2 < .01$, and $F_{\text{Syn}}(4, 284) = 2.10$, $p = .08$, $\eta_g^2 = .01$, that is, there was no evidence for an influence of identifier style at this relatively coarse level of analysis.

5.3.2 First Pass Reading Time (FPRT)

We operationalized *first pass reading time* as the time it took participants to navigate the letterbox from entering the actual code after the comments section (i.e., $AOI_{\text{Pre-Defect}}$) to the position where defect first appeared in the letterbox (i.e., AOI_{Defect}). Some participants briefly screened the whole code snippet before reading by quickly moving from top to bottom. We therefore only considered passes that took at least five seconds.

Table 9 shows an increased first pass reading time for the single-letter identifier names in the semantic task. The interaction term of a repeated-measures ANOVA with identifier naming style and task type as independent variables supports a selective effect of identifier naming style, depending task type, $F(2, 142) = 4.45$, $p = .01$, $\eta_g^2 = .01$. A corresponding simple effects analysis shows that, as predicted, there was a statistically significant effect in the semantic condition, $F_{\text{Sem}}(2, 142) = 7.60$, $p < .001$, $\eta_g^2 = .05$, but not in the syntactic condition, $F_{\text{Syn}}(4, 142) = 0.24$, $p = .79$, $\eta_g^2 < .01$.

5.3.3 AOI visits

We expected that in the semantic task single letter identifier names and abbreviations would force participants to revisit the comments section more frequently to refresh their memory of what the identifier names denote. To test this hypothesis, we calculated how often each AOI was visited by participants, that is, how often they entered an AOI (coming from a different AOI) and stayed for at least one second, without moving the letterbox. We introduced this constraint eliminate the effect of merely scrolling through an AOI. The number of visits

Table 9 First-pass reading times by identifier naming style in seconds

	Semantic		Syntactic	
Style	*Median*	*IQR*	*Median*	*IQR*
Word	23.62	22.63	15.38	12.52
Abbreviation	27.45	25.79	16.54	14.75
Letter	35.50	28.62	16.97	11.95

Table 10 AOI visit counts by AOI and identifier naming style

		Semantic		Syntactic	
AOI	Style	*Median*	*IQR*	*Median*	*IQR*
Comment	Word	1.73	1.34	1.00	0.64
	Abbreviation	1.93	1.37	1.09	0.67
	Letter	2.19	2.14	1.05	0.58
Pre-Defect	Word	2.29	1.97	1.24	1.08
	Abbreviation	2.29	2.43	1.26	1.21
	Letter	2.57	2.64	1.19	0.83
Defect	Word	2.09	1.77	1.26	1.10
	Abbreviation	2.00	1.72	1.19	1.02
	Letter	1.98	2.29	1.15	0.81

are displayed in Table 10. The statistical analysis confirmed a significant interaction of identifier naming style and task type for the number of visits to the comments section, $F(2, 142) = 5.02$, $p = .008$, $\eta_g^2 = 0.02$. A follow-up analysis showed that the effect was clearly present in the semantic condition, $F_{\text{Sem}}(2, 142) = 9.13$, $p < .001$, $\eta_g^2 = 0.06$, but not in the syntactic condition, $F_{\text{Syn}}(4, 142) = 2.73$, $p = .07$, $\eta_g^2 = 0.02$.

6 Discussion

To summarize the results, our data show that participants found semantic defects significantly faster when the code presented used normal words as identifier names compared to both abbreviations and meaningless single letters. Furthermore, finding syntax errors appears to be unaffected by identifier naming style. These results indicate that program comprehension benefits from explicit identifier names, as comprehension of code with words as identifier names was 19% faster compared to abbreviations and letters.

Although the *word-length effect* predicts that people can keep short strings in memory more easily, single letters and abbreviations did not lead to an improvement of comprehension speed. Instead, longer words seemed to facilitate comprehension, as semantic defects were discovered faster when words were used as identifier names. Moreover, contrasting the semantic task with the syntactic task shows that purely perceptual properties of identifier names, such as their length, are also insufficient to explain performance differences when the task does not require semantic judgments. However, in this case the word identifiers' semantic content is not a benefit either. The observed differences in comprehension performance are likely caused by the words' semantic content, which facilitates forming an appropriate mental model of how the code operates and generally relieves working memory. This is supported by the fact that in the word condition participants scrolled less frequently to the comments of the source code to retrieve the meaning of a variable. In other words, word identifier names allow developers to access the meaning of a concept represented by an identifier directly, which may allow them to reason about the code more easily.

In the present study, the letterbox allowed us to investigate the process of reading code to some extent. Although we found no effect of identifier naming style on overall dwell time (i.e., how long participants spent reading particular parts of comments or code), our analyses of visual focus revealed that naming style had an effect on how often participants returned to the comments section and how long the first pass of reading the code took. Single letter identifiers resulted in the slowest first-pass reading time and the highest number of visits to the comments section. This further supports the assumption that code containing single letter identifiers is read less fluently and leads to more frequent failures to recall the meaning of identifiers.

Our data indicate that in line with $RH_{Semantic}$, words as identifier names lead to faster detection of semantic defects. When in-depth understanding was not required, as in the task to find syntax errors, the identifier names did not make a difference, supporting RH_{Syntax}. Considering these results, arguments in favor of non-word identifiers seem questionable. The disadvantage of increased typing effort of longer words seems to be outweighed by the benefits of their stronger semantics, especially when considering that code is more often read than written. In practice, this drawback could be diminished with appropriate tooling. For example, modern IDEs provide auto-completion facilities, which already reduce typing effort. In future studies, it would be interesting to quantify the relationship between typing effort in modern IDEs and comprehensibility.

7 Threats to validity

7.1 Internal validity

We used only 53% of all completed data records, which could be interpreted as a sampling bias, but this was in fact a result of our strict filtering rules that were applied to improve the quality of our data by reducing the effect of nuisance factors unrelated to the goal of the experiment (e.g., language barriers, distractions).

Program comprehension as a construct is difficult to measure. We had reasoned that code can only be corrected when a person has fully understood how the code works and therefore operationalized program comprehension using defect finding tasks. However, reading code to understand it may not be the the same as reading code that needs to be fixed and thus it is possible that our task carries some overhead. For example, identifying that something "is a car" is arguably an easier task than to identify the reason why it will not start, but the former process is required to enable the latter. Similarly, identifying that a code snippet sorts an array is a different task than to find out the complexity of the algorithm, or whether or not it contains a defective corner-case. Participants might have responded more quickly in a different task (e.g., read until you feel you understood the code), but with the chosen design we ensured that participants had understood the code by evaluating the correctness of their responses.

Compared to a video-based eye-tracker, the letterbox method for measuring visual focus has some limitations. It enforces sequential scrolling, thus slowing down the reading process and requiring to scroll through the middle parts of the code when moving from top to bottom or vice versa. This constraint does not distort the comparison between conditions, as it was constant across conditions. However, it reduces the explanatory power of the visual focus data, in particular, the uncorrected dwell-times, because every scrolling movement counts

towards this measure. The AOI Visits were less influenced by this effect, because they correct for scrolling.

7.2 External validity

For practical reasons we limited our sampling to a specific population, namely German professional C# developers. While linguistic or cultural differences could have an effect, we assume that professional software developers in different countries share many characteristics. Furthermore, code and comments were written in English which should further enhance the generalizability of findings. Only two participants identified as female in the initial raw data, but their records were not included in the final dataset after applying exclusion criteria. However, program comprehension is a complex cognitive activity and we expect that professional education and experience would outweigh potential gender differences.

Our stimulus materials were limited to procedural, algorithmic problems and therefore do not allow us to draw conclusions about the impact of identifier naming styles in complex, object-oriented environments, where identifiers such as `AbstractSingletonProxyFactoryBean` can be common. These very long identifier names try to be explicit, but they might be too abstract to provide meaningful semantics to facilitate program comprehension, thus hindering the performance of program comprehension. Although compound identifier names comprising more than one word may convey meaningful details, we had disregarded such identifier names because they would have been inappropriate in the scope of our code snippets. Furthermore, word identifiers can also be rendered meaningless if they are unrelated to the purpose of the entities they designate. For example, when loop variables are labeled `superman` and `batman`, both variables have a distinct semantic content, but this content is unrelated to the designated purposes. Effects of this type need to be investigated in following experiments.

We had used very short and meaningless single letters identifier names, but even single letters can be considered meaningful under certain circumstances. For example, using `i` as a running variable in a loop can be considered a widely accepted convention. Other conventions and habits could positively affect program comprehension, such as domain-specific identifier names that have been established as conventions in their specific domain. For example, a system dealing with linear-algebra, such as a drawing engine, may commonly use identifier names such as `x`, `y`, and `z` to denote Cartesian coordinates. Although these letters are very short, their semantics are clear and using different identifiers other than `x`, `y`, and `z` could even slow down programmers trying to comprehend the code.

For our experimental setup, we chose tasks that were relevant to developers' daily work (i.e., finding semantic defects), but in order to reach out to participants, these tasks were performed online, rather than in an IDE, and thus might be regarded as artificial. Our web-application might have slowed down comprehension performance by means of the restricted focus viewer, and comprehension should be faster under normal circumstances. However, because this impediment was present to all participants in all experimental conditions, it should not have distorted the relative effects of different identifier naming styles.

Finally, it should be noted that modern, complex code bases can have millions of lines of code. In our experiment, participants were 19% faster in comprehending 15 lines of code with word identifier names. If this comprehension advantage scales up to larger code bases, this could save hours or even weeks of time required to comprehend code. However, it is difficult to predict this with certainty. It is possible that common abbreviations in a larger code base are memorized quickly, and developers do not suffer a penalty during program comprehension.

8 Related work

Lawrie et al. (2006) performed a similar experiment in which participants read code with letters, abbreviations, and words. They found that responses for identifier names using abbreviations were similar to responses for normal words, whereas in our experiment they were more similar to responses for letters. We attribute this difference to the strategies used for building abbreviations. Lawrie et al. (2006) abbreviated longer composite identifier names rather than single words (e.g., `isPrimeNumber` to `isPriNum`). Such abbreviations retain more similarity to the original identifier name compared to the identifiers used in our study. We therefore see the results by Lawrie et al. (2006) in accordance with ours.

In a subsequent paper, Lawrie et al. found indications that identifier name length interacts with working memory to such an extent that words and abbreviations are easier to identify in recognition tasks than single letters (Lawrie et al. 2007). Again, this applies to the longer abbreviations in their study. This further confirms our results that semantics are relevant for the comprehension of source code and that purely perceptual explanations of differences in comprehension performance are insufficient.

Although we found that semantic properties are more important for program comprehension than low-level perceptual properties of identifier names, the research on identifier-splitting techniques shows that both aspects should be considered as complementary. Syntactic properties of identifier names may facilitate perceptual processes, whereas semantic properties facilitate higher level cognitive processes. Both aspects should be considered to write code that people can understand optimally.

The studies by Sharif and Maletic (2010) and Binkley et al. (2009) found that better comprehension was achieved when participants were presented with code that was congruent with their previous experiences. The study by Binkley et al. showed that participants who were experienced with `camelCase` took less time to identify `camelCase` identifiers compared to `under_score` identifiers. Aligning their own results with these findings, Sharif and Maletic conclude that "with more experience (training), the effect of identifier style on performance is reduced, but not eliminated" (Sharif and Maletic 2010, p. 203). Thus, experience seems to play a role when determining relevant factors of program comprehension. Our data exhibit similar characteristics. We found that the observed effects (the impact of identifier naming styles) reside in the middle and the tail of the distribution of reaction time data. This indicates that *experts* (i.e., the fastest developers in our sample) were less influenced by shorter or abbreviated identifier names than developers with average performance.

Ceccato et al. (2014) analyzed different code-obfuscation techniques that intentionally make code difficult to comprehend. They could show that renaming identifiers to single characters is an effective obfuscation technique to hinder program comprehension, although it does not render it impossible. These results underline the importance of good identifier naming styles, as comprehending code seems to be easier when words are used and impeded when letters are present.

Scalabrino et al. (2016) found that using textual properties of source code, including semantic aspects, such as coherence and narrowness of identifier names, improve the prediction of *readability* over and above using structural aspects such as line lengths, number of identifiers, or number of parentheses. In the work of Scalabrino, as well as other code readability studies (e.g., Buse and Weimer 2010; Posnett et al. 2011), readability is often operationalized as participants' subjective judgments whether or not a snippet is readable. In contrast to subjective ratings such as these, our experiment employed response times as a behavioral performance measure. Ideally, subjective and objective measures of readability

will lead to converging results, but this relationship should be investigated further to clearly establish the validity of either construct.

9 Conclusion

Given that maintenance and program comprehension play a crucial role in software development (most likely more than typing code), it seems advisable to use explicit full-word identifiers. Our results indicate that abbreviations and letters reduce a program's comprehensibility, and their presence might be an indicator for lower quality code.

We could show that shorter identifier names are not necessarily better for program comprehension. Semantics play an important role during comprehension in general, thus using words as identifier names may facilitate the comprehension of computer programs. Developers should optimize their code to support cognitive processes dedicated to interpreting word semantics by using explicit identifier names. Using longer identifier names is made easy by code completion features of modern IDEs and additional typing effort is minimal.

To aid comprehension, appropriate rules for style guides should consider perceptual and semantic properties and discourage the use of ad-hoc abbreviations and single letters. Instead, they should encourage the use of explicit, clear identifier names given that they wish to improve the quality of software and reduce its development and maintenance costs.

In future work, it would be interesting to investigate how longer, more complex identifier names behave, for example, compound identifier names in larger programs. Furthermore, it remains an open question to what extent the effect of identifier naming style depends on general and domain specific programming experience. In general, we welcome replications of the results presented to evaluate their robustness in different contexts. For this purpose, we provide a replication package with all required materials on the project website.

Acknowledgements This work has been supported by the DFG grant SI 2045/2-1. Janet Siegmund's work is further funded by the Bavarian State Ministry of Education, Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B).

Compliance with Ethical Standards This study was performed in accordance with the ethical standards of the Department of Psychology, Heidelberg University, Germany.

Conflict of interests The authors declare that they have no conflict of interest.

References

Anquetil N, Lethbridge T (1998) Assessing the relevance of identifier names in a legacy software system. In: Conf. centre for advanced studies on collaborative research, CASCON '98. IBM Press, Toronto, pp 1–10

Baddeley AD, Thomson N, Buchanan M (1975) Word length and the structure of short-term memory. J Verbal Learn Verbal Behav 14(6):575–589. https://doi.org/10.1016/S0022-5371(75)80045-4

Bakeman R (2005) Recommended effect size statistics for repeated measures designs. Behav Res Methods 37(3):379–384. https://doi.org/10.3758/BF03192707

Balota DA, Chumbley JI (1985) The locus of word-frequency effects in the pronunciation task: lexical access and/or production? J Mem Lang 24(1):89–106. https://doi.org/10.1016/0749-596X(85)90017-8

Binkley D, Davis M, Lawrie D, Morrell C (2009) To CamelCase or under_score. In: Proc. Int'l conf. program comprehension (ICPC), pp 158–167. https://doi.org/10.1109/ICPC.2009.5090039

Brooks R (1983) Towards a theory of the comprehension of computer programs. Intl J Man-Mach Stud 18(6):543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

Buse RPL, Weimer WR (2010) Learning a metric for code readability. IEEE Trans Softw Eng (TSE) 36(4):546–558. https://doi.org/10.1109/TSE.2009.70
Ceccato M, Di Penta M, Falcarin P, Ricca F, Torchiano M, Tonella P (2014) A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empir Softw Eng 19:1040–1074
Cohen J (1988) Statistical power analysis for the behavioral sciences. Erlbaum, Hillsdale
Collins AM, Loftus EF (1975) A spreading-activation theory of semantic processing. Psychol Rev 82(6):407–428. https://doi.org/10.1037/0033-295X.82.6.407
Coltheart M, Rastle K, Perry C, Langdon R, Ziegler J (2001) DRC: a dual route cascaded model of visual word recognition and reading aloud. Psychol Rev 108(1):204–256
Cowan N (2001) The magical number 4 in short-term memory: a reconsideration of mental storage capacity. Behav Brain Sci 24(1):87–185
Deissenboeck F, Pizka M (2006) Concise and consistent naming. Softw Qual Control 14(3):261–282. https://doi.org/10.1007/s11219-006-9219-1
Hofmeister J, Siegmund J, Holt DV (2017) Shorter identifier names take longer to comprehend. In: 2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER), pp 217–227. https://doi.org/10.1109/SANER.2017.7884623
Jansen AR, Blackwell AF, Marriott K (2003) A tool for tracking visual attention: the restricted focus viewer. Behav Res Methods Instrum Comput 35(1):57–69
Lawrie D, Morrell C, Feild H, Binkley D (2006) What's in a name? A study of identifiers. In: Proc. Int'l conf. program comprehension (ICPC), pp 3–12. https://doi.org/10.1109/ICPC.2006.51
Lawrie D, Morrell C, Feild H, Binkley D (2007) Effective identifier names for comprehension and memory. Innov Syst Softw Eng 3(4):303–318. https://doi.org/10.1007/s11334-007-0031-2
Leonhart R (2009) Lehrbuch Statistik Einstieg und Vertiefung, 2nd edn. Hans Huber, Hogrefe AG, Bern
Miller GA (1994) The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychol Rev 101(2):343–352
MSDN (2016) Class naming guidelines [online]. available: https://msdn.microsoft.com/en-us/library/4xhs4564(v=vs.71).aspx
Posnett D, Hindle A, Devanbu P (2011) A simpler model of software readability, ACM, New York
Ratcliff R (1993) Methods for dealing with reaction time outliers. Psychol Bull 114(3):510–532
Scalabrino S, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2016) Improving code readability models with textual features. In: Proc. Int'l conf. program comprehension (ICPC), pp 1–10. https://doi.org/10.1109/ICPC.2016.7503707
Sharif B, Maletic JI (2010) An eye tracking study on camelcase and under_score identifier styles. In: Proc. Int'l Conf. program comprehension (ICPC). Proc. Int'l Conf. Program Comprehension (ICPC). IEEE Computer Society, Washington, DC, pp 196–205. https://doi.org/10.1109/ICPC.2010.41
Sneed H (1996) Object-oriented COBOL Recycling. In: Proceedings of the Third working conference on reverse engineering, 1996, pp 169–178. https://doi.org/10.1109/WCRE.1996.558901
Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. IEEE Trans Softw Eng SE 10(5):595–609. https://doi.org/10.1109/TSE.1984.5010283
Tichy WF (1998) Should computer scientists experiment more? In: IEEE Computer
Weekes BS (1997) Differential effects of number of letters on word and nonword naming latency. Q J Exper Psychol Sec A 50(2):439–456. https://doi.org/10.1080/713755710
Whelan R (2008) Effective analysis of reaction time data. Psychol Record 58(3):475

Johannes C. Hofmeister works as a software developer and research associate at the University of Passau, Germany. His research focuses on how people express themselves in code and the cognitive processes involved in program comprehension.

Janet Siegmund is currently working at the University of Passau, where she is leading the junior research group PICCARD, funded by the Centre Digitisation.Bavaria. She received her Ph.D. from the University of Magdeburg in 2012 and she holds two master's degrees, one in Computer Science and one in Psychology. In her research, she focuses on the human factor in software engineering, for example, when writing source code.

Daniel V. Holt is a lecturer in Theoretical and Cognitive Psychology at the Department of Psychology, Heidelberg University, Germany. His research interests include executive functions, self-regulation and programming from a problem solving perspective.