Effects of Variable Names on Comprehension: An Empirical Study

Eran Avidan   Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

Abstract—**It is widely accepted that meaningful variable names are important for comprehension. We conducted a controlled experiment in which 9 professional developers try to understand 6 methods from production util classes, either with the original variable names or with names replaced by meaningless single letters. Results show that parameter names are more significant for comprehension than local variables. But, surprisingly, we also found that in 3 of the methods there were no significant differences between the control and experimental groups, due to poor and even misleading variable names. These disturbingly common bad names reflect the subjective nature of naming, and highlight the need for additional research on how variable names are interpreted and how better names can be chosen.**

Index Terms—**code comprehension; variable names; misleading names; method parameters; local variables**

I. Introduction

Code comprehension is a task present in many aspects of a programmer's daily work. In particular, it is pivotal in facilitating effective software maintenance and enabling successful evolution of computer systems [27]. As Martin writes, "the ratio of time spent reading vs. writing is well over 10:1. We are constantly reading old code as part of the effort to write new code" [17]. This implies the need to understand code that was written by another programmer or in the past. Likewise, examples are important when learning skills such as using new frameworks or languages [26], as witnessed by the burgeoning use of sites like Stack Overflow. And again, one must understand the code examples to benefit from them.

An important element of code comprehension is to understand the underlying concepts embodied in the code [22]. In principle, such concepts should be described by in-code documentation and design documents [21]. But documentation is often missing or outdated. So in many cases the most important and trusted beacons[1] that provide signals about concepts are identifiers. It has even been said that "if a name requires a comment, then the name does not reveal its intent" [17], suggesting identifiers *are* the documentation.

Identifiers are also prevalent. In large open source projects about a third of the tokens are identifiers, and they account for about two thirds of the characters in the source code [8]. It is therefore doubly important that they convey meaning. The importance of meaningful identifier names was established in a number of papers [6], [24]. For example, Lawrie et al. found that full word identifiers and abbreviations may lead to better comprehension than identifiers composed of single letters [16]. This was done using controlled experiments, where the different experimental treatments were versions of the same methods but with different identifier names. Our work continues this line of research using a similar methodology, but focusing on the identifiers' meaning rather than their length.

As may be expected, not all identifiers are born equal: some are more important than others. This has been reflected in naming recommendations, such as the adage that "The length of a name should correspond to the size of its scope" [17]. In particular, several authors have expressed the belief that variables in method headers, namely the method's parameters, are more important than local variables. Our goal is to test such beliefs empirically. Specifically, we aim to examine the impact of local variables and parameters names by themselves on the comprehension process of software developers approaching an unseen snippet of Java code.

Our contributions to the field are

1) To quantify the effect of meaningful names in an experimentally valid manner (one-on-one experiments with industry professionals in their work environment with no dropouts, and using real methods from popular production codes).
2) To demonstrate that bad names that actually impede comprehension are not uncommon, illustrating the fact that naming is subjective, and motivating placing an emphasis on names in code reviews.
3) To identify parameters as more important than locals in most cases. This is part of the whole signature being important, with implications to API design.
4) On the methodological level, to demonstrate the use of experiments with dynamic treatments, where the treatment changes during the experiment.

II. Background on Identifier Naming

Practically all programming guidelines state that variables should be given "meaningful names". But naming is hard. Furnas et al. studied spontaneous word choice

[1] In program comprehension the term "beacon" describes a code element which illuminates the code's function beyond this element's immediate use.

for objects in different application-related domains, and found the variability to be surprisingly high: in every case two people favored the same term with a probability of less than 0.2 [12]. Arnaoudova et al. studied identifier renaming, and found that the vast majority of cases include changes or at least modifications of meaning [4]. Problematic situations include mismatch of type, number, or behavior (e.g. a `set` method that returns) [3].

Hindle et al. show that in large projects the vast majority of the vocabulary used is identifiers, and that the vocabularies of different projects tend to be more diverse than the commonly used vocabulary in natural language [15]. Rilling and Klemola have shown that code fragments with a high identifier density may act as "comprehension bottlenecks" [23]. These results imply that identifier naming is not self-evident, and that programmers need guidance when it comes to naming identifiers in their code.

Regrettably typical coding conventions supply very superficial guidelines on naming, focusing on style and formatting, with no regard to their meaning and what they represent. For instance, "Method names are written in lowerCamelCase" and "Package names are all lowercase" are very common conventions. The *Java code conventions* do mention the importance of meaning, saying "Variable names should be short yet meaningful" and "designed to indicate to the casual observer the intent of its use" [25]. This is a good start, but hardly enough, considering that different people give different names to the same thing.

How can one write more useful guidelines? Binkley et al. have suggested rules to improve field names based on natural language processing providing part-of-speech information [5]. Deißenböck and Pizka created a formal model with naming rules that check consistency, conciseness, and composition of each element name [8]. Caprile and Tonella are more practical, and suggest standardization of variable names using a lexicon of concepts and syntactic rules for arranging them [7]. Allamanis et al. have recently presented a framework that learns the style of a codebase, building on recent work in applying statistical natural language processing to source code [1]. This was then applied to suggest natural identifier names and formatting conventions and to suggest revisions to improve stylistic consistency. Unfortunately, it is not clear that any of the above ideas is used in practice.

Gellenbeck et al. found that both meaningful procedure and variable names serve as beacons to high-level comprehension [13]. Similarly, Osman et al. found that names are crucial for the understanding of UML diagrams — without them there is no clue of what the different classes actually do [20]. Haiduc et al. found that when summarizing code, developers tend to include in the summary practically all the terms that appear in method names, and the vast majority of terms in parameter types [14]. This suggests that they perceive these elements as conveying important information regarding what the method does.

Our study stresses the importance of proper identifier naming, and shows the impact that unsatisfactory names have on comprehension. We also use controlled experiments to show that some variables have a higher impact on comprehension than others, and thus should be given more attention. To the best of our knowledge this distinction has not been made explicitly before.

III. Research Questions

Our specific research questions are as follows:

1) *What impact do identifier names have on the comprehension of what a method does?* This is essentially a replication of previous work, perhaps with some methodological variation.
2) Assuming names are important, *which type of identifiers contribute more to comprehension, parameters or local variables?* This is a new distinction that has not been studied before. It is interesting because locals with limited scope have been disparaged in coding guidelines as not necessarily requiring meaningful names, whereas parameters are part of a method's signature and thus potentially part of an API. But do these different roles compel different levels of naming?

In the experiments these questions are treated together, by comparing methods that either have meaningful names or else replace some or all of the names by single letters.

IV. Research Design

A. Controlled Experiments

An important category of empirical study is the controlled experiment, which is the classical scientific method for identifying cause-effect relationships. Our experiment was designed to assess the effect of variable names on comprehension, using real methods from utility packages. To remove meaning from identifier names we replace them with consecutive letters of the alphabet. This allows compilation, but conveys absolutely no information. In particular, it facilitates the generation of code where either parameters *or* locals, or even both, are devoid of meaning.

Experimental subjects were professional developers working at a major hi-tech company in Israel. The experiments were conducted by the first author in multiple individual sessions with each subject. In each session subjects were presented with the task of understanding one or more methods, in either an experimental treatment or a control treatment. Overall 38 sessions were recorded, totaling approximately 22 hours.

B. Experimental Treatments

The experimental treatments are versions of the selected methods with different classes of variable names masked out. In addition the method name was removed, replacing it with 'xxx', to enable a focus on variable names and avoid any confounding effect (including both possibilities: that the method name aids comprehension or that it is misleading). From each method we prepared four versions:

```
public static void xxx(final boolean[] a,
                 final int b, final int c) {
    if (a == null) {
        return;
    }
    int d = b < 0 ? 0 : b;
    int e = Math.min(a.length, c) - 1;
    boolean f;
    while (e > d) {
        f = a[e];
        a[e] = a[d];
        a[d] = f;
        e--;
        d++;
    }
}
```

Fig. 1: Method with all variables replaced by single letters (version 4).

1) Only the method name was removed. All variable names remained intact.
2) The method name was removed and *parameters* were replaced with single letters.
3) The method name was removed and *local variables* were replaced with single letters.
4) The name was removed and *both* parameters *and* local variables were replaced with single letters.

When variable names were replaced by single letters, these were `a`, `b`, `c`, and so on in order of appearance. An example of a method with all names replaced by single letters is shown in Figure 1.

The **control treatment** consisted of using version no. 1, namely the original code as is except for the method name. This retains all the information that was embedded in the variable names by the original programmers.

The **experimental treatment** consisted of using 3 versions in sequence. Initially we presented the participant with version 4 of the method, where all variables were replaced with single letter identifiers in alphabetical order. The participant was asked to explain what is the method's purpose. Once an answer was received, we asked how sure is he in this answer, and if he feels certain enough and would like to move forward and receive one more type of identifier names. This phase was identical in all the experimental treatment sessions.

We then revealed either the parameters or the local variables identifier names (essentially switching to version 3 or 2, respectively). The decision of which to reveal was random. The participant was asked whether his understanding had changed or his confidence improved due to this additional information. Finally, we revealed the other type of identifier names, leading to code that has all the original variable names (version 1). This is the same in all sessions, and identical to the control treatment.

This dynamic change of treatment was used to reduce the number of subjects and experiments required, and the length of each session, which was up to thirty five minutes even so. In addition, it enabled an observation of the "aha" moment when variables are revealed and it makes an immediate difference to the comprehension and/or confidence.

All the participants went through all three phases. Each phase was limited to ten minutes: after ten minutes the subject was asked what he believes the method does and his confidence level, after which he was presented with one more type of identifier names.

C. *Experimental Design*

Naturally each subject who performs an experiment with the control treatment for a certain method cannot also perform the experimental treatment, and vice versa, because the method is already familiar. We are therefore forced to use a "between subjects" design when comparing versions of the same method.

We randomly divided the subjects into two groups, S1 and S2, and the methods into two groups, M1 and M2. Group S1 did methods M1 under control conditions, and methods M2 under experimental treatment. Group S2 did the opposite: methods M1 under experimental treatment and methods M2 as control.

D. *Variables*

The main independent variables are the treatments, and naturally also the subjects (and their experience, sex, etc.) and the methods.

The main dependent variable is time for comprehension. This variable represents the time it took for each participant to give a correct answer. For the experimental treatment the time was measured from the beginning of phase one, when the participants were first presented with the striped method, up until the time the right answer was received. This time is probably a lower-bound on the time required to understand the method without any variable names, as variables are gradually revealed. Deciding that a participant correctly comprehends a method was based on using the think-aloud methodology with interactions. Thus to ascertain comprehension the experimenter read the participant's description back to him, and asked followup questions to verify meaning. The observer never confirmed to the participant the correctness of his answer, and the experiment continued until phase three, even if the time measurement had stopped because a correct answer had already been given.

Another dependent variable is each subject's subjective opinion of identifier type importance, namely whether parameters or locals were more important for their understanding of the method. This is naturally influenced by the stage in which they reached an understanding.

E. *Code Selection*

Context plays a major role in code comprehension, which may lead to a confounding effect: lack of understanding may result from bad identifier names (or other aspects of code complexity) or from lack of domain knowledge. Domain knowledge also impacts the way programmers approach the code [19]. In order to eliminate the need for context and domain knowledge, we chose to use methods from popular open source utility packages. In addition to making the code accessible, this facilitates using real code that was developed by different programmers.

Searching for classes from which to extract methods was done as follows. First, we reviewed the most popular Java repositories on github, identified as those starred by at least 10K users. Then the nature of the repository was evaluated, to understand the potential for finding robust utility classes. For those repositories which seemed promising, a manual examination of the source code was used to select suitable methods. During this process we reviewed over 30 different repositories and over 200 different classes.

The selected packages were Apache Commons, Google Guava, and Spring Framework. We chose util classes for data types that should be familiar to any Java developer, arrays and strings. Choosing the methods to use in the experiments was harder than expected. Many methods are just too trivial, or contain beacons that make it easy to simply guess their purpose. Likewise, we avoided methods that use uncommon types or programming styles, in order to keep the focus on identifier naming and not on design patterns or coding techniques. Finally, we made an effort to choose diverse methods, in order to eliminate a learning process that may lead to successful guesses instead of code comprehension.

Initially we extracted 12 suitable methods with 10–30 lines of code and 3–10 parameters and local variables. We then conducted a pilot with two subjects, one performing the control treatment and the other an experimental treatment. Based on the pilot we removed some candidate methods that were found to be too hard or easy relative to other ones. These were mainly either short methods that contain very few identifiers, as the lack of variables made them inappropriate for our study, or long methods with many identifiers that were simply too hard to follow. We ended up with the 6 methods characterized in Table I, where their descriptions are taken from the online Java documentation of the packages. More details and code will be presented when we discuss the results.

Naturally the choice of specific methods has a subjective element. Reproducibility is however guaranteed as anyone can use the same methods we chose. Alternatively, other researchers can apply the same considerations and select their own methods. This is exactly the type of variation that leads to better confidence in experimental results (or to uncovering differences that need to be investigated) [11].

F. *Participant Recruitment*

The subjects who took part in this study were all industry professionals from Israel. They all work at the same division of a major hi-tech company and spend most of their time in code development. We allowed participants with different tasks, project roles, and experience in order to explore program comprehension as broadly as possible and to improve external validity. According to [16] men may understand single-letter abbreviations better than women. Despite the fact that our single letter identifiers are not abbreviations, in order to eliminate this threat to validity, the recruitment was limited to men.

The recruitment was done personally by the first author, starting with random selection from a longer list of developers. In total there were 9 participants in the experiment: 6 developers, a team leader, a senior developer, and a technical lead. The age of participants ranged from 25 to 45, and work experience from 3 to 20 years. All participants' work experience includes a good knowledge of Java. Eight of the subjects participated in sessions on all 6 methods, but in one case one session was discarded from the analysis due to interference during the session. One subject was hard to schedule and did only one session.

The experiments were conducted during working hours in the participants' working place, and did not require any investment beyond attending the conference room for the duration of the sessions. The participants showed significant interest in the study, and seemed to appreciate the importance of identifier naming and the lack of sufficient guidelines. Consequently, they all gladly volunteered to participate without getting any kind of compensation.

G. *Experimental Procedure*

Each subject participated in multiple sessions scheduled on different days. Control sessions included 2–3 different methods, and experimental sessions one method. At the beginning of each session the subject was given a short reminder of the purpose and process of the current session. We explained that he will be presented with a "real world" method with its name removed, and that we would like to know what this method is meant to do. In experimental treatment sessions we also explained that whenever he feels satisfied with his answer we would reveal one more type of identifier names. Based on the pilot, participants were asked to think aloud to give us a better understanding of how they understand the code [10]. We concluded with questions on what the method name should be and whether local variables or parameters were more beneficial for comprehension.

Note that we use an unorthodox experimental procedure with these two features:

1) Dynamic change of code version as the subject makes progress — we start with no variable names, and then add the names of parameters and locals in a random order. Thus our experimental treatments are actually a sequence of 3 versions.
2) Some level of interaction between the experimenter and the subject, as asking questions adjacent to actions is expected to result in the most direct and "capture the moment" answers.

TABLE I: The methods which were used in the experiment.

ID	*Name*	*LOC*	*Params*	*Vars*	*Description*
1	reverse	15	3	3	Reverses the order of the given array in the given range
2	indexOfAny	27	2	5	Find the first index of any of a set of potential sub-strings
3	substringsBetween	30	3	5	Searches a String for sub-strings delimited by a start and end tag, returning all matching substrings in an array
4	replaceChars	28	3	6	Replaces multiple characters in a String in one go
5	repeat	25	2	5	Repeat a String repeat times to form a new String
6	abbriviateMiddle	20	3	4	Abbreviates a String to the length passed, replacing the middle characters with the supplied replacement String

The experiments were conducted in front of a laptop in a quiet room, with only the experimenter and subject present. Based on the pilot, subjects were provided pen and paper to enable simulating the execution of the code to aid comprehension. They could not run the code. For documentation we kept a protocol of every experimental session, including a webcam video, screen capture, observer notes, and the participant's notes and code alterations if any.

H. Statistical Methods

Our study is based on comparing small samples sizes, which most likely are not normally distributed. This prevents us from using the common t-test. Instead, we chose to use the Mann-Whitney U test [18]. This is a nonparametric test of the null hypothesis that two independent samples A and B come from the same distribution, against the alternative that one population A tends to have larger values than another population B (in other words, that A stochastically dominates B). More formally, in the specific case of our experiments the null hypothesis is

> H_0: the time it takes a programmer that receives the experimental treatment to understand a method has the same distribution as the time it takes a programmer that receives the control treatment

The results were that for some methods we were able to reject the null hypothesis at a significance level of 0.05.

Since we are comparing two small sets of observations, the calculation of the U statistic is trivial. First, assign numeric ranks to all the observations, beginning with 1 for the shortest time for comprehension. Then add up the ranks for the observations which came from the control group (which we denote R_c) and from the experimental group (R_e). Note that $R_c + R_e = N(N+1)/2$ which is the sum of all ranks up to N, the total number of observations. Now calculate $U_i = R_i - n_i(n_i+1)/2$ where $i \in \{c, e\}$ and n_i is the number of observations in the control and experimental groups, respectively. Finally compare $\min(U_c, U_e)$ to a standard table of Mann-Whitney critical values and reject or accept the hypothesis accordingly.

I. Validity Concerns and Mitigation

Some decisions leading to the experimental procedure described above were taken explicitly to mitigate threats to validity. A major concern was establishing construct validity, meaning that we really measure comprehension. This led to the decision to focus on real production code that is unlikely to be known to subjects, as opposed to using textbook examples and algorithms which might be recognized. Production code also contributes to external validity as it better represents the code developers encounter in their daily work. Another issue with construct validity is the possible confounding effect of lack of domain knowledge. This was mitigated by using util classes.

Another concern is that masking the method names creates an unrealistic comprehension scenario, because in real life method names would be available. Moreover, method names are expected to provide significant information. But this is precisely why they need to be masked in order to enable the study of the effects of variable names. Moreover, method names can also be misleading, adding a confounding effect.

Another external validity issue is the possible use of students as subjects, as was done in many previous studies on program comprehension. We preferred professionals because students normally hardly ever read code, whereas professionals need to read and comprehend code a lot [17]. We conjecture that as a result professionals develop expertise in comprehension of unknown code, making them especially suited for this specific type of experiment. Moreover, the recruitment of professional developers in their workplace increases the experimental realism, and, thereby, the applicability of the results.

The most common internal validity threat is that individual differences between experimental subjects may mask the measured effects. We mitigated this concern by having each subject participate in both experimental and control treatments. Choosing same sex subjects from the same company prevented unintended confounding effects of sex and work culture.

V. Results and Analysis

In retrospect, the results obtained with the 6 methods we used exhibit interesting diversity. We therefore start by describing the results for each method in detail, and then proceed to summarize and discuss the observed effects.

A. reverse

The `reverse` method is shown in Figure 2 (and version 4 with all variable names masked was shown previously in Figure 1). It reverses the part of an array between two indexes. Cumulative distribution functions of the times for

```
public static void xxx(final boolean[] array,
      final int startIndexInclusive,
      final int endIndexExclusive) {
   if (array == null) {
       return;
   }
   int i = startIndexInclusive<0 ? 0 : startIndexInclusive;
   int j = Math.min(array.length, endIndexExclusive) - 1;
   boolean tmp;
   while (j > i) {
       tmp = array[j];
       array[j] = array[i];
       array[i] = tmp;
       j--;
       i++;
   }
}
```

Fig. 2: `reverse` method.

```
public static int xxx(final CharSequence str,
      final CharSequence... searchStrs) {
   // some lines not shown
   final int sz = searchStrs.length;
   int ret = Integer.MAX_VALUE;
   int tmp = 0;
   for (int i = 0; i < sz; i++) {
       final CharSequence search = searchStrs[i];
       if (search == null) {
           continue;
       }
       tmp = CharSequenceUtils.indexOf(str, search, 0);
       if (tmp == INDEX_NOT_FOUND) {
           continue;
       }
       if (tmp < ret) {
           ret = tmp;
       }
   }
   return ret == Integer.MAX_VALUE ? INDEX_NOT_FOUND : ret;
}
```

Fig. 3: `indexOfAny` method (some lines removed).

comprehension are shown in Figure 4. Participants under the control treatment understood the method's functionality in 2.5–8 minutes, while those with the experimental treatment took 7–19 minutes to come up with a sufficient answer. Due to the separation between the time ranges in the control and experimental treatments, calculating the Mann-Whitney statistic returned a U-value of 1. This leads to the conclusion that the time for comprehension in the control group was significantly lower than in the experimental group, with a significance level of 0.05.

Five out of six experimental participants indicated that the parameters were more beneficial to their comprehension. Moreover, while three of them were given the parameter names first and three the local variables, all six reached the correct answer only after the parameters were revealed. Participants who were given the local variables first were quick to request the parameters as well, saying that the local variables had very little value to them.

Interestingly, all of them missed the fact that the end index is exclusive, despite the "-1" in line 8 that suggests this, until the parameter `endIndexExclusive` was revealed. One stated in disappointment "I missed the -1 in the end exclusive index", and another said "from the start I saw the -1 but did not treat it right". This demonstrates how beacons in identifier names can focus attention on related code.

All participants described the functionality of the method correctly as a series of actions, but none of them used phrases such as "reversing the array" which describe it at a higher level of abstraction. But when asked to name the method, one participant did call it `reverseRange`.

B. *indexOfAny*

`indexOfAny` is shown in Figure 3. This method finds the index of the first of a set of potential substrings. Participants under the control treatment understood the method functionality in 2–7 minutes, while those of the experimental treatment took 12.5–22 minutes (figure 4). Due to the complete separation between the time ranges the Mann-Whitney statistic was $U = 0$, again leading to the conclusion that time to comprehension in the control group was lower than in the experimental one with a significance level of 0.05.

All 4 experimental participants indicated that the parameters were more beneficial to their comprehension. Moreover, all of them reached the correct answer only after receiving the parameter names, regardless of the order names were revealed. We also noticed the rise in the level of confidence as more identifiers were revealed, even if this did not prompt the subjects to change their answer.

Two participants in the experimental treatment completely missed the '...' type annotation in the second parameter. One was even recorded saying "ahhh there are the three dots, was it there from the beginning? I could have known it before if I noticed the annotation." The other two seemed to notice it, but still treated the parameter as a single `charSequence`. A possible reason is that variable arguments with the '...' notation are not commonly used, so its significance was not obvious. Two participants described the `for` loop as "running char by char" when pointing to the line with `final CharSequence g = b[f];` (the replacement for line 7), even though `g` is clearly of type `charSequence` and not `char`. It may be that the single letter name `b` gave the impression of a single object, in this case a string, and grasping the fact that it represents more than one object did not come naturally.

None of the participants gave the method the same name as its developer. One did give a similar name, `indexOfFirstOccurrence`.

C. *subStringsBetween*

This is a longer method with 30 lines of code which searches a string for all occurrences of substrings delimited by given start and end tags (Figure 5). The participants with the control treatment took 6–15 minutes to reach a

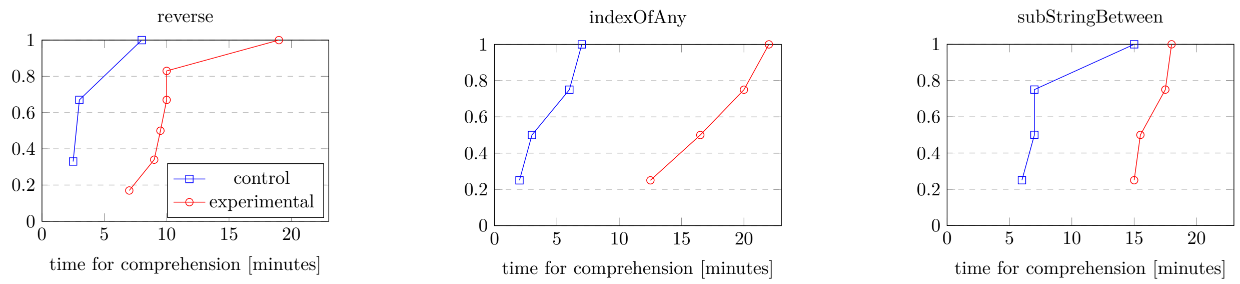

Fig. 4: Cumulative distribution functions of required time in the control treatment vs. the experimental treatment.

```
public static String[] xxx(final String str,
        final String open, final String close) {
    // some lines not shown
    final int closeLen = close.length();
    final int openLen = open.length();
    final List<String> list = new ArrayList<String>();
    int pos = 0;
    while (pos < strLen - closeLen) {
        int start = str.indexOf(open, pos);
        if (start < 0) {
            break;
        }
        start += openLen;
        final int end = str.indexOf(close, start);
        if (end < 0) {
            break;
        }
        list.add(str.substring(start, end));
        pos = end + closeLen;
    }
    if (list.isEmpty()) {
        return null;
    }
    return list.toArray(new String [list.size()]);
}
```

Fig. 5: `subStringsBetween` method (some lines removed).

correct answer, but the distribution indicates only one of them really found it hard to follow the code (Figure 4). The participants with the experimental treatment gave a correct answer in 15–18 minutes, and all of them resorted to a pen and paper in the process. As in the previous two cases, the Mann-Whitney test indicated that the control treatment time was lower than the experimental time with a significance level of 0.05.

All participants reported that identifiers containing “open” and “close” were very informative, so receiving them as parameters (`open`, `close`) or as locals (`openLen`, `closeLen`) was the most helpful clue in understanding the method. As a result all of the participants reached their answer after the first type of identifier was revealed, be it parameters or locals.

D. *replaceChars*

With this method we observed an interesting phenomenon: 2 out of 4 control participants gave the same *incorrect* answer, while all of the experimental treatment participants reached the correct answer. The method

```
String xxx(final String str,
    final String searchChars, String replaceChars)
```

Fig. 6: `replaceChars` method header

header can be seen in Figure 6. What it actually does is to replace multiple characters in a string in one go, by searching for characters in the `searchChars` parameter string, and replacing them with the corresponding characters in `replaceChars`. In the first few minutes 3 of the 4 controls mistakenly claimed that the functionality was quite clear from the parameters alone, and they only needed to verify their intuition that “it performs a simple replace string”. One changed his answer after further examining the code, while the other two remained with their initial intuition. Though it took the control group less time than the experimental one, 4.5–10 minutes vs. 8–12 minutes (Figure 7), this partial separation in the time for comprehension is meaningless since half of the answers of the control group were incorrect. Therefore no statistical test was performed.

Inputs from the control group indicate that the main reason for their mistake was that `searchChars` and `replaceChars` were perceived as a search string and a replace string. This may be attributed to two root causes: 1) Lack of language sensitivity. To non-English natives, “chars” and “string” can look like synonyms. 2) The fact that the type of these parameters is `String` conflicts with their use as a collection of chars. Hence, a better option, which was also suggested by some participants, would be to change the type to an array of `chars`. In any case, it appears that just as identifiers can inform and speed up comprehension, they can also hinder its correctness, by supplying beacons for a different mental model than intended. And when one gets an intuition regarding a method’s functionality, it is hard to change it, even when some beacons point to a different solution.

E. *repeat*

This method repeats a string several times to form a new string (Figure 8). It uses a non-trivial system function, `arrayCopy`, which led all participants in the experimental treatment to read the attached Java documentation. On the other hand, none of the control group felt a need to read the documentation, and they did not seem to

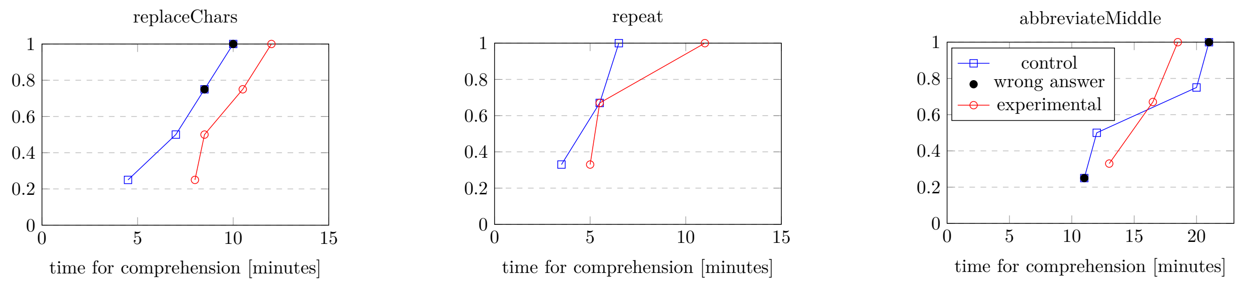

Fig. 7: Cumulative distribution functions of required time in the control treatment vs. the experimental treatment.

```
public static String xxx(String string, int count) {
    // some lines not shown
    final int len = string.length();
    final long longSize = (long) len * (long) count;
    final int size = (int) longSize;
    // some lines not shown
    for (n = len; n < size - n; n <<= 1) {
        System.arraycopy(array, 0, array, n, n);
    }
    System.arraycopy(array, 0, array, n, size - n);
    return new String(array);
}
```

Fig. 8: `repeat` method (some lines removed).

```
public static String xxx(final String str,
     final String middle, final int length) {
  // some lines not shown
  final int targetSting = length-middle.length();
  final int startOffset = targetSting/2+targetSting%2;
  final int endOffset = str.length()-targetSting/2;
  final StringBuilder builder = new StringBuilder(length);
  builder.append(str.substring(0,startOffset));
  builder.append(middle);
  builder.append(str.substring(endOffset));
  return builder.toString();
}
```

Fig. 9: `abbreviateMiddle` method (some lines removed). `targetSting` typo in the source.

pay much attention to the function at all. This suggests that good identifiers not only aid the comprehension of a specific method, but that the lack of good names could lead to the need to comprehend additional methods.

The control group reached a correct answer in 3.5–6.5 minutes, while the experimental group needed 5–11 minutes, with no significant separation (Figure 7). The control group seemed to get their understanding from the parameters, while the experimental subjects got the intuition from identifying the patterns where the returned string size is the original string's length times `count`. However, they were a bit hesitant with their answer, starting with "in my opinion" and a confidence level of 60%, or "my guess is" with 80%.

Moreover, participants mentioned that some of the names are confusing. Specifically, `size` and `len` have practically the same meaning, but one pertains to the input and the other to the output. `count` is imprecise: its role is to state the number of times to repeat and not to count. Finally, `string` is a redundant name for a String type variable, and can be overlooked as String the type when scanning the code.

F. *abbreviateMiddle*

This method abbreviates a String to the given length, by exchanging its middle with the provided replacement string (Figure 9). It is relatively short, but all subjects needed to verify every line in order to understand what it does. We attribute this to the fact that its functionality is very unique. In fact, it was hard for the subjects to grasp that there is a method that does such a thing, they did not understand "why would someone write a method for that?" Evidently, when the apparent functionality does not make sense, it is harder to comprehend the code.

Moreover, the variable names were not meaningful and even misleading. Perhaps the most offending one is the int `targetString`. The name implies a string, but the type is an integer, and the actual usage is to calculate the length of those parts in the original string that will be retained in the output string. So this name is obviously misleading; a better name could be something like `lengthToCopy`. Likewise, the parameter `length` is too general, and participants asked "length of what?", which led them to all kinds of directions. A more appropriate name might have been `targetLength`.

Presumably due to the bad names, the subjects who received the control treatment seemed to be more confused. One of them gave up after 21 minutes, and another reached a wrong answer. The experimental subjects took about the same time and all reached correct answers (Figure 7).

VI. Discussion

Given the above results, we now summarize their implications for the research questions, as well as unexpected observations such as the detrimental effect of bad names.

A. *The importance of meaningful names*

The results for the first three methods (`reverse`, `indexOfAny`, and `substringsBetween`) show a clear separation between treatments. In all three methods the time it took participants to reach comprehension under the control treatment was shorter than the time needed with the experimental treatment, where variable names were replaced by a, b, c, and so on. This result was statistically significant at a level of 0.05 using the Mann-Whitney test. It conforms with previous results in the literature that also showed names to be important [6], [16], [24].

The flip side of meaningful names is misleading names. The `replaceChars` and `abbreviateMiddle` methods provide striking examples. With these methods participants who received the experimental treatment were found to perform better despite the missing identifier names! Moreover, participants in the control treatment, who saw the full original identifier names, tended to make mistakes and arrive at wrong conclusions. Obviously, bad names can mislead just as much as good names inform. This is not a deep observation, but our results do demonstrate and quantify the effect using real production code and professional developers.

In another method, `repeat`, variables were not so good, but not as bad as the misleading variables noted above. In addition, two methods suffered from a clash between identifier names and their type. In `replaceChars` this was a mismatch between the type `String` and the name `chars`. In `indexOfAny` hiding the plural name led to masking of the variable parameters notation.

Importantly, we did not select these methods to demonstrate bad names, and in fact we did not realize how bad their variable names were when we started the experiment. The fact that half of the methods we selected turned out to have problematic names is a warning that such names are most probably not uncommon. This may explain why results reported in the literature are sometimes inconsistent with each other.

B. Parameters vs. Locals

Our experiments indicate that parameters contribute more to the code comprehension than locals: they were deemed more beneficial by the participants in 79% of the cases. Figure 10 shows the distribution of answers for the different methods. In some cases, notably `reverse`, `indexOfAny`, and `abbreviateMiddle`, parameters were nearly universally preferred. This is notable as it implies that local variables alone were not enough, and the participants really needed the parameters in order to understand these methods. We believe this can have several reasons:

- Parameter names are more carefully chosen by the original developers. Developers may even tend to embed comprehension beacons in parameters because they are part of the interface that defines the method.
- Because parameters are part of the header, they might facilitate a more top-down approach to comprehension. As one subject phrased it: “Parameters are more beneficial, since once I have a clear starting point everything else is easier”.
- Missing information about local names can be compensated by seeing how they are computed.

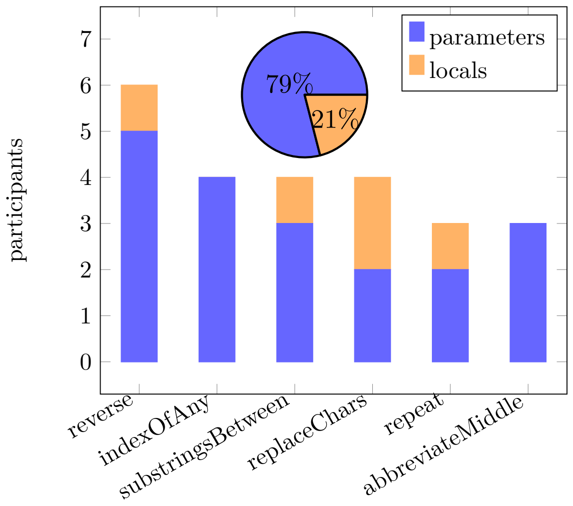

Fig. 10: parameters vs. locals: which was claimed to be more significant for comprehension.

However, in some cases it was not the parameters that mattered, but which identifiers were revealed first. This was especially prominent in `substringsBetween` and `replaceChars`, where the first variables to be revealed—either parameters or locals—were always sufficient for comprehension. This can happen when the main beacon for comprehension appeared in *both* the parameters and the local variables. It suggests that at least in some cases it’s not the location or type of variables but their actual name that makes the difference. Indeed, in the majority of cases where participants chose the local variables as more beneficial, the main beacon was embedded in the parameters as well.

VII. Limitations and Threats to Validity

As noted above, several decisions about the experimental design were taken specifically to mitigate threats to validity. However, other threats remain.

Construct validity refers to correctly measuring the dependent variable, in our case the time to understand a method. A possible risk stems from our experimental procedure which allows a dialog between the observer and subject. This requires that the observer will restrain himself from pointing the subject to the solution or misleading him away from it. So, there is the risk that the observer will affect the comprehension process by mistake.

Another possible problem is the dynamic treatment where new information is revealed along the way. This is expected to assist in the comprehension process, but it also requires flexible thinking to assimilate the new information [9]. Individuals who are less flexible may suffer from the new information.

Internal validity refers to causation: are changes in the dependent variable necessarily the result of manipulations to treatments? A possible problem in our work is that sometimes local variable names may mirror parameter names. For instance, given a parameter called `str`, a method may include a local variable called `strLength` for its length. As a result the separation of parameters from locals is compromised: even if we remove parameter names, the information leaks via the local name. In addition, information may leak due to the use of library functions. These problems seem unavoidable when using real code.

Another risk is that parameters may have been found to be more important because, being part of the API, they more directly correlate with the question of "what does the method do?". Moreover, the attribution of effect to variables may be too strong, as in the experiments we masked the method comments and the method name. In real life programmers can benefit from these additional beacons, and the effect of variable names will be reduced.

External validity refers to generalization. Conducting the experiment on a small number of methods, performing actions on arrays and strings, may not be generally representative. Moreover, the fact that the authors of this paper chose those methods contains the risk of bias. Still, we did use 6 different methods and observed a range of behaviors. The recruitment of only male subjects working at the same company, though intentional, may lead to lack of generalization outside the company and to female developers.

VIII. Future Work

Our choice to replace the method names with "xxx" makes the study less realistic, as names are naturally available when developers attempt to understand a method. Moreover, it mixes the understanding of a method's functionality (in other words, its "contract") with the understanding of its code (the contract's implementation). Therefore two variations of the current experiment should be performed to complete the picture. First, we would like to conduct experiments with the original method names in place, to examine the effect of the method name relative to the variables. These experiments thus focus on the API level. Second, we want to replace the "what does the method do?" question with a more code-related task like debugging or refactoring. Such experiments focus on the implementation level. Taken together, these experiments are more realistic as they have a better match of the experimental conditions and the task.

Another problem with our design is that developers usually do not focus exclusively on a single method, and the patterns in which methods call each other naturally convey information. A big challenge is therefore to improve the experiments' realism by increasing the scope from a single method to a class or package.

Finally, all these experiments will benefit from being reproduced using other code and different experimental subjects (specifically including women). There are most probably myriad subtle effects at play, e.g. the quality of names, which are hard to quantify and characterize. The cumulative work of multiple researchers is needed to achieve more comprehensive results.

IX. Conclusions

Replicating previous work on identifier naming reconfirms that names have a large impact on the comprehension of code. Good names can effectively serve as the code's documentation, and are instrumental for comprehension. We also showed that method parameter names are typically more significant for comprehension than local variable names. This can be because parameter names are more carefully chosen due to their part in the API, and thus in the definition of the method. Alternatively, it can just be due to their location at the top of the method.

Surprisingly, three of the six methods we used turned out to have problematic names that even led to comprehension errors. These demonstrate that misleading names, or names that clash with their types, are worse than meaningless names like consecutive letters of the alphabet. But these names were not meant to be misleading. This reflects the subjective nature of naming, where a name that one developer thinks is meaningful can be misleading for another developer.

The main implication of our work is that names must be picked with caution and given careful attention, so they reflect the concept or role represented by each variable. But different people have different mindsets and backgrounds, which may lead to misunderstandings. Thus it would be interesting to map additional examples of misleading names, and try to identify common traits that make them bad. We leave this to future work.

Extending this, our results suggest practitioners would benefit from including the issue of naming in code reviews, by making sure that all participants interpret the names in the same way. Current practice is that code reviews follow extensive checklists of what to look for, but none of these checklists address naming problems. Moreover, bad names are not considered to be defects and do not figure in defect reports. They should.

More generally, our work highlights the need for additional research on how names are interpreted and how better names can be chosen. An ambitious goal would be to devise tools capable of effective automatic evaluation and suggestion of meaningful names. Recent work has exhibited progress on this front, using techniques from natural language processing and machine learning [1], [2]. Time will tell whether this can indeed compensate for the subjective nature of naming.

Verifiability

All versions of the methods used in this study are available at `http://bit.ly/2bbosZL`.

Acknowledgments

This study was made possible by its volunteer participants. Thank you all. This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

References

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "*Learning natural coding conventions*". In 22nd *Foundations Softw. Eng.*, pp. 281–293, Nov 2014, DOI:10.1145/2635868.2635883.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "*Suggesting accurate method and class names*". In 10th *Joint ESEC/FSE*, pp. 38–49, Sep 2015, DOI:10.1145/2786805.2786849.

[3] V. Arnaoudova, M. Di Penta, and G. Antoniol, "*Linguistic antipatterns: What they are and how developers perceive them*". *Empirical Softw. Eng.* **21(1)**, pp. 104–158, Feb 2016, DOI: 10.1007/s10664-014-9350-8.

[4] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Guéhéneuc, "*REPENT: Analyzing the nature of identifier renamings*". *IEEE Trans. Softw. Eng.* **40(5)**, pp. 502–532, May 2014, DOI:10.1109/TSE.2014.2312942.

[5] D. Binkley, M. Hearn, and D. Lawrie, "*Improving identifier informativeness using part of speech information*". In 8th *Working Conf. Mining Softw. Repositories*, pp. 203–206, May 2011, DOI:10.1145/1985441.1985471.

[6] S. Blinman and A. Cockburn, "*Program comprehension: Investigating the effects of naming style and documentation*". In 6th *Australasian User Interface Conf.*, pp. 73–78, Jan 2005.

[7] B. Caprile and P. Tonella, "*Restructuring program identifier names*". In *Intl. Conf. Softw. Maintenance*, pp. 97–107, Oct 2000, DOI:10.1109/ICSM.2000.883022.

[8] F. Deißenböck and M. Pizka, "*Concise and consistent naming*". In 13th *IEEE Intl. Workshop Program Comprehension*, pp. 97–106, May 2005, DOI:10.1109/WPC.2005.14.

[9] C. G. DeYoung, J. B. Peterson, and D. M. Higgins, "*Sources of openness/intellect: Cognitive and neuropsychological correlates of the fifth factor of personality*". *Journal of personality* **73(4)**, pp. 825–858, 2005.

[10] K. A. Ericsson and H. A. Simon, *Protocol analysis.* MIT press Cambridge, MA, 1993.

[11] D. G. Feitelson, "*From repeatability to reproducibility and corroboration*". *Operating Syst. Rev.* **49(1)**, pp. 3–11, Jan 2015, DOI:10.1145/2723872.2723875.

[12] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "*The vocabulary problem in human-system communication*". *Comm. ACM* **30(11)**, pp. 964–971, 1987, DOI: 10.1145/32206.32212.

[13] E. M. Gellenbeck and C. R. Cook, "*An investigation of procedure and variable names as beacons during program comprehension*". In 4th *Workshop on Empirical Studies of Programmers*, pp. 65–79, 1991.

[14] S. Haiduc, J. Aponte, and A. Marcus, "*Supporting program comprehension with source code summarization*". In 32nd *Intl. Conf. Softw. Eng.*, vol. 2, pp. 223–226, 2010, DOI: 10.1145/1810295.1810335.

[15] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "*On the naturalness of software*". *Comm. ACM* **59(5)**, pp. 122–131, May 2016, DOI:10.1145/2902362.

[16] D. Lawrie, C. Morrell, H. Field, and D. Binkley, "*What's in a name? a study of identifiers*". In 14th *Intl. Conf. Program Comprehension*, pp. 3–12, Jun 2006, DOI:10.1109/ICPC.2006.51.

[17] R. C. Martin, *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[18] N. Nachar, "*The mann-whitney U: A test for assessing whether two independent samples come from the same distribution*". *Tutorials in Quantitative Methods for Psychology* **4(1)**, pp. 13–20, 2008.

[19] M. P. O'Brien and J. Buckley, "*Inference-based and expectation-based processing in program comprehension*". In 9th *IEEE Intl. Workshop Program Comprehension*, pp. 71–78, 2001.

[20] H. Osman, A. van Zadelhoff, D. R. Stikkolorum, and M. R. V. Chaudron, "*UML class diagram simplification: What is in the developer's mind?*" In 2nd *Workshop Empirical Studies Softw. Modeling*, art. no. 5, Oct 2012, DOI:10.1145/2424563.2424570.

[21] D. L. Parnas and P. C. Clements, "*A rational design process: How and why to fake it*". *IEEE Trans. Softw. Eng.* **SE-12(2)**, pp. 251–257, Feb 1986.

[22] V. Rajlich and N. Wilde, "*The role of concepts in program comprehension*". In 10th *IEEE Intl. Workshop Program Comprehension*, pp. 271–278, Jun 2002, DOI:10.1109/WPC.2002.1021348.

[23] J. Rilling and T. Klemola, "*Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics*". In 11th *IEEE Intl. Workshop Program Comprehension*, pp. 115–124, May 2003.

[24] F. Salviulo and G. Scanniello, "*Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals*". In 18th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. no. 48, May 2014, DOI: 10.1145/2601248.2601251.

[25] R. Stoll, "*Type-safe PHP: Java code conventions*" 2014.

[26] K. VanLehn, "*Cognitive skill acquisition*". *Ann. Rev. Psychol.* **47**, pp. 513–539, 1996, DOI:10.1146/annurev.psych.47.1.513.

[27] A. von Mayrhauser and A. M. Vans, "*Program comprehension during software maintenance and evolution*". *Computer* **28(8)**, pp. 44–55, Aug 1995, DOI:10.1109/2.402076.