The Hebrew University of Jerusalem

Faculty of Science

The Rachel and Selim Benin School of Computer Science and Engineering

The Significance of Method Parameters and Local Variables as Beacons for Comprehension: An Empirical Study

Author: **Eran Avidan**

Supervisor: **Prof. Dror G Feitelson**

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science

November 2016

אני פסיק על ידה

I would first like to thank my thesis advisor Prof. Dror G Feitelson. The door to his office was (literately) always open.

I would also like to thank all the developers who volunteered to take part in this research. Without their passionate participation and input, the experiments could not have been successfully conducted. You made this study possible. Thank you all.

Finally, I must express my very profound gratitude to my dear wife for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without you.

Abstract

It is widely accepted that meaningful variable names are important for comprehension. We claim that not all variables are equally significant to the process of code 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. In our experiments, we have masked all the methods names. One group of subjects was presented with 3 different versions of the code in the following order: all variable names replaced by single letters, only parameters names or local variables replaced by single letters, and all the original variable names. We then compared the time it took them to resolved the method functionality to a different group who received the method with the original variable names revealed from the beginning. During the experiments we communicated with the subjects and used the thinking aloud technique in order to get a better understanding of each developer's thinking process. This helped us determine, among other things, which of the variables were more beneficial to their understanding of the methods.

Results show that parameter names are more significant for comprehension then local variables, as 80% of the subjects indicated that parameters where more beneficial then locals regardless of the order the variable names were revealed. Replicating previous work on identifier naming, we can also reconfirms that names have a large impact on the comprehension of code. In half of the methods there was a statistically significant difference between the time it took each group to reach an understanding of the method, were it took less time for the subjects who were presented with the method and its original variable names.

Surprisingly, the other three of the six methods we used turned out to have problematic names. In fact, they demonstrated that misleading names are worse than meaningless names like consecutive letters of the alphabet, and can lead to errors in comprehension. 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. Considering the way these methods were chosen, and the fact that they were extracted from real production code, suggests that it is a not uncommon phenomenon. We believe that this highlights the need for additional research on how variable names are interpreted and how better names can be chosen.

Contents

Chapter 1

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 [29]. 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" [18]. This implies the need to access and comprehend code that was written by another programmer or even by the same programmer in the past. Likewise, examples are important teaching tools when learning new frameworks or languages, as witnessed by the burgeoning use of sites like Stack Overflow. Research in cognitive science confirms that "examples appear to play a central role in the early phases of cognitive skill acquisition" [28]. And again, one must understand the code examples to benefit from them.

So what are the enablers of comprehension? A basic prerequisite for understandability is readability. The basic syntactical elements must be easy to spot and easy to recognize. Only then, one can establish relationships between the elements and gain comprehension. But this is obviously not enough. An important element of code comprehension is to understand the underlying concepts embodied in the code [23]. In principle, such concepts should be described by in-code documentation and design documents [22]. But documentation is often missing or outdated. So in many cases the best available beacons 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" [18], 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 [6]. It is therefore doubly important that they convey meaning. The importance of meaningful identifier names was established in a number of papers [4,26]. For example, Lawrie et al. found that full word identifiers and abbreviations may lead to better comprehension than identifiers composed of single letters [17]. 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.

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" [18]. In particular, several authors have expressed the belief that single letter variables should be limited to local scope, e.g. in the form of loop indexes. Our goal is to test such beliefs empirically. Specifically, we distinguish between variables that have distinct roles in a method: those that are parameters of the method and those that are local variables. Given this distinction, we ask which variables are more important for comprehension: parameters or locals?

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). Interestingly, a couple of the methods we used ended up demonstrating the ill effects of bad variable names.

2. 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.

3. On the methodological level, to demonstrate the use of experiments

with dynamic treatments, where the treatment changes during the experiment.

1.1 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 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].

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 [16]. Rilling and Klemola have shown that code fragments with a high identifier density may act as "comprehension bottlenecks" [25]. 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" [27]. This is a good start, but hardly enough, considering that different people give different names to the same thing.

Another example can be found in the Google Java Style Guide [14] introduction which states that it "focuses primarily on the hard-and-fast rules that we follow universally, and avoids giving advice that isn't clearly enforceable (whether by human or tool)." This of course leads to lack of sufficient guidelines for identifier naming, since, as mentioned before, they are not so clearly enforceable.

How can one extend such guidelines and make them more useful? Binkley

et al. have suggested rules to improve field names based on natural language processing providing part-of-speech information [3]. Deißenböck and Pizka created a formal model with naming rules that check consistency, conciseness, and composition of each element name [6]. Caprile and Tonella are more practical, and suggest standardization of variable names using a lexicon of concepts and syntactic rules for arranging them [5]. Raychev et al. have presented a new approach for predicting variable names from a massive code base using machine learning [24], which could decrease the variety of names given to the same concept. 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 [21]. 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 [15]. 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 the comprehension process. We also use controlled experiments to show that some variables have a higher impact on comprehension, and should be given more attention than others. To the best of our knowledge this distinction has not been made explicitly before.

Chapter 2

Research Questions

The specific research questions that underlie our work are as follows:

1. *What impact does the lack of meaningful identifier names have on the comprehension of a method?* This is essentially a replication of previous work, perhaps with some methodological variation. It is the basis for our work because if meaningful identifier names are not important the second question is irrelevant.

2. *Which type of identifiers contribute more to the comprehension process, local variables or parameters?* 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 addition to these top-level questions we also consider issues such as generality, namely whether the results pertain to all code or perhaps different circumstances may lead to different results. In the experiments all these research questions are treated together, by comparing methods that either have meaningful names or replace some or all of the names by single letters.

Chapter 3

Research Design

To remove meaning from identifier names we use consecutive letters of the alphabet. They allow compilation, but convey absolutely no information, so they figure prominently in our experimental methodology. In particular, this facilitates the generation of code where either parameters *or* locals, but not both, are devoid of meaning.

3.1 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 but with the method name removed. Experimental subjects were professional developers working at a major hi-tech company in Israel. The experiments were conducted by the 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.

3.2 Experimental Treatments

The goal of the experiment is to assess the effect of identifier names on program comprehension. Subjects were presented with methods selected from popular open-source utility packages, with variable names potentially replaced by single letters. In addition the method name was removed to avoid any confounding effect (including both possibilities: that the method name aids comprehension or that it is misleading). The procedure for selecting the methods is described below.

3.2.1 Method versions

From each selected method we prepared four versions:

1. Only the method name was removed, replacing it with 'xxx'. All variable names remained intact.
2. The name was removed and *parameters* replaced with single letters.
3. The name was removed and *local variables* replaced with single letters.
4. The name was removed and *both* parameters *and* local variables 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 3.1.

3.2.2 Control treatment

The control treatment consists of using version no. 1, namely the original code as is except for the method name which is replaced by 'xxx'. This represents retaining all the information that was embodied in the variable names by the original programmers.

```
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++;
    }
}
```

Figure 3.1: Method with all variables replaced by single letters (version 4).

3.2.3 Experimental treatment

The experimental treatment consisted of using 3 versions in sequence.

Initially we presented the participant with version no. 4 of the method, where the name was replaced with 'xxx', and both parameters and local variables were replaced with single letter identifiers in alphabetical order as seen in Figure 3.1. 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 assignment 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, in the third phase we reveal 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. In total, each session took up to thirty five minutes.

3.3 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.

3.4 Variables

The independent variables are naturally the subjects, the methods, and the treatments. In addition one may consider implied variables such as the subject's years of experience.

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. 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. This time is probably a lower-bound on the time required to understand the method without any variable names, as variables are gradually revealed.

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.

3.5 Code Selection

Context plays a major role in the code comprehension process. When measuring code comprehension this 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 [20]. In order to eliminate the need for context and domain knowledge, we chose to use methods from popular open source utilities packages. This actually leads to two benefits: first, it makes the code more accessible to all developers, and second, it facilitates using "real" code that was developed by different programmers.

Searching for classes from which to extract methods was done in the following way. First, we reviewed the most popular Java repositories on github, where "popular" is one which is known to be frequently used or is stared 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 specific methods to use in the experiments was challenging. While going through all the methods the following considerations were applied. Many methods are just too trivial, or contain beacons that make it easy to simply guess their purpose. We also excluded short methods that contain very few identifiers, as the lack of variables made them inappropriate for our study, and long methods with many identifiers that were simply too hard to follow. 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 choosing diverse methods, in order to eliminate a learning process that may lead to successful guesses instead of code comprehension.

In total, we extracted 12 suitable methods from the above repositories. Each method had 10–30 lines of code and contained 3–10 parameters and local variables. From each candidate method we prepared versions reflecting the different treatments as described above. 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. We ended up with 6 methods for use in the experiments. These methods are characterized in Table 3.1, 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) [10].

Table 3.1: 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

3.6 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. The age of participants ranged from 25 to 45, and work experience from 3 to 20 years. According to [17] 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 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. 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 experimentation was 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. Interestingly, the participants showed increased interest in the study, stressing the importance of identifier naming and insufficient guidelines and consistency for them. Consequently, they all gladly volunteered to participate without getting any kind of compensation.

3.7 Experimental Procedure

Each subject participated in multiple sessions. 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 think and understand the code [9]. We concluded with questions on what he believes the method name should be and what was more beneficial for his comprehension process, the local variables or parameters identifier names.

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.

The experiments were conducted in front of a laptop in a quite 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. 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.

3.8 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, which performs well on small samples [19]. 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 control treatment to understand a method has the same distribution as the time it takes a programmer that receives the experimental treatment

In the results section we show that for some methods we are 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.

3.9 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 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 [18]. 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.

Chapter 4

Results and Analysis

In retrospect, the results obtained with the 6 methods we used exhibit some interesting diversity. We therefore start by describing the results for each method in detail. The next section then proceeds to summarize and discuss the observed effects regarding the importance of identifier names and parameters vs. locals.

4.1 reverse

The `reverse` method is shown in Figure 4.1 (and version 4 with the parameters and local variables replaced by single letters was shown previously in Figure 3.1). It reverses the part of an array between two indexes. Cumulative distribution function of the times for comprehension can be seen in Figure 4.2. Participants who received the control treatment understood the method's functionality in 2.5–8 minutes, while for those who received the experimental treatment it 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 identifiers were more beneficial to their comprehension process. Moreover,

```
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++;
    }
}
```

Figure 4.1: `reverse` method (version 1).

while three of them were given the parameter names first and three the local variables, all six reached the correct answer only after the parameter names were revealed. Participants who were given the local variable names first were quick to request the parameter names 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, even though there is a "-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 that "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`. Suggested method names for all the methods used in the experiment can be found in Table 4.1

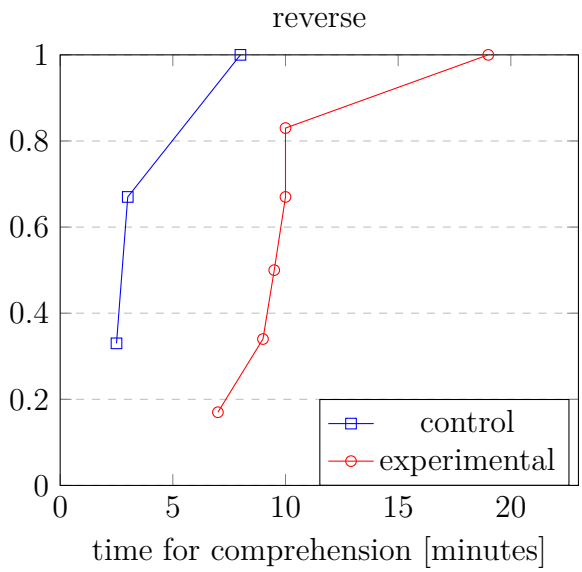

Figure 4.2: Commutative distribution function of required time in the control treatment vs. the experimental treatment.

4.2 indexOfAny

`indexOfAny` is shown in Figure 4.3. This method finds the index of the first of a set of potential substrings. Participants who received the control treatment understood the method functionality in 2–7 minutes, while those who received the experimental treatment took 12.5–22 minutes (figure 4.4). Due to the complete separation between the time ranges calculating the Mann-Whitney statistic returned a U-value of 0, which again led us to conclude that the time for comprehension in the control group was significantly lower than the one 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 process. Moreover, all of them reached the correct answer only after receiving the parameter names, regardless of the order names were revealed to them. 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.

Half of the participants in the experimental treatment completely missed the '...' type annotation in the second parameter. One was even recorded say-

```java
public static int xxx(final CharSequence str,
    final CharSequence... searchStrs) {
    if (str == null || searchStrs == null) {
        return INDEX_NOT_FOUND;
    }
    final int sz = searchStrs.length;

    // String's can't have a MAX_VALUEth index.
    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;
}
```

Figure 4.3: `indexOfAny` method (version 1).

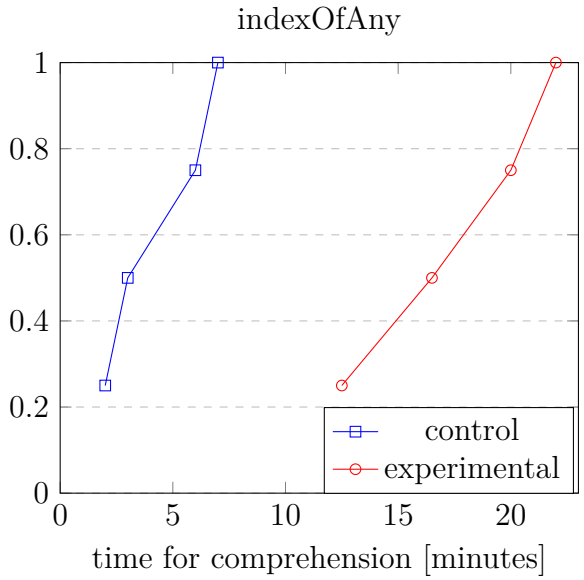

Figure 4.4: Commutative distribution function of required time in the control treatment vs. the experimental treatment.

ing "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 three dots notation are not commonly used by developers, so it's 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];` (where `b` is the replacement for `searchStrs`), even though `g` is clearly of type `charSequences` 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`.

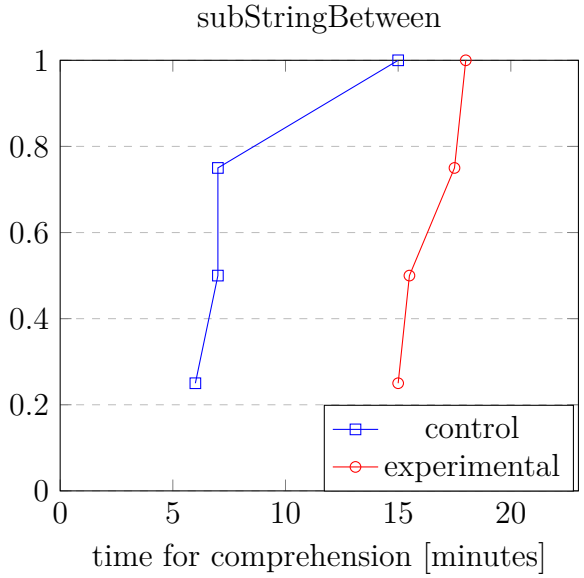

Figure 4.5: Commutative distribution function of required time in the control treatment vs. the experimental treatment.

4.3 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 4.6). The participants who received the control treatment took 6–15 minutes to reach a correct answer, so at least for one of them the number of variables and the length made it hard to follow the code even with its original names (Figure 4.5). The participants with the experimental treatment gave a correct answer in 15–18 minutes. 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.

It seems that the relatively higher number of identifiers had an impact on the comprehension process, requiring much to be remembered. As a result, all of the experimental treatment participants resorted to a pen and paper.

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

```java
public static String[] xxx(final String str, final String open,
    final String close) {
    if (str == null || isEmpty(open) || isEmpty(close)) {
        return null;
    }
    final int strLen = str.length();
    if (strLen == 0) {
        return ArrayUtils.EMPTY_STRING_ARRAY;
    }
    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()]);
}
```

Figure 4.6: `subStringsBetween` method (version 1).

method. As a result all of the participants reached their answer after the first type of identifier was revealed, be it parameters or locals. One even went as far as to say that from the parameters alone he could have probably understand the method functionality.

4.4 replaceChars

Using this method we observed an interesting phenomenon: 2 out of 4 participants who received the control treatment gave the same *incorrect* answer, while all of the experimental treatment participants reached the correct answer. In the first few minutes 3 out of the 4 participants 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", which was incorrect. 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, this partial separation in the time for comprehension is meaningless since half of the answers supplied by the control treatment group were incorrect (Figure 4.8). Therefore no statistical test was performed.

The method can be seen in Figure 4.7. What it actually does is to replace multiple characters in a string in one go. It does so by searching for characters in the `searchChars` parameter string, and replacing them with the corresponding characters in `replaceChars`.

Inputs from the control group lead us to believe the main issue that led to 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) The native language barrier. To non-English natives, "chars" and "string" can seem like synonyms. 2) The fact that the type of these parameters is `String` suggests that they will be used as a string and not 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 speed up the comprehension process, they can also hinder its correctness, by supplying beacons for a different mental

```java
public static String xxx(final String str, final String searchChars,
   String replaceChars) {
   if (isEmpty(str) || isEmpty(searchChars)) {
       return str;
   }
   if (replaceChars == null) {
       replaceChars = EMPTY;
   }
   boolean modified = false;
   final int replaceCharsLength = replaceChars.length();
   final int strLength = str.length();
   final StringBuilder buf = new StringBuilder(strLength);
   for (int i = 0; i < strLength; i++) {
       final char ch = str.charAt(i);
       final int index = searchChars.indexOf(ch);
       if (index >= 0) {
           modified = true;
           if (index < replaceCharsLength) {
               buf.append(replaceChars.charAt(index));
           }
       } else {
           buf.append(ch);
       }
   }
   if (modified) {
       return buf.toString();
   }
   return str;
}
```

Figure 4.7: `replaceChars` method (version 1)

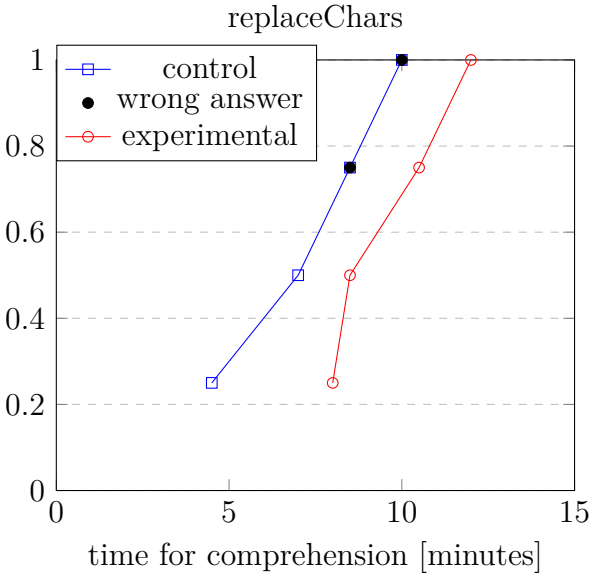

Figure 4.8: Commutative distribution function of required time in the control treatment vs. the experimental treatment.

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.

4.5 repeat

This method repeats a string several times to form a new string (Figure 4.9). 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 the need to use the Java documentation, and didn't seem to pay much attention to the function at all. This suggests that good identifiers not only impact the comprehension process of a specific method, but that the lack of good names could lead to the need to comprehend additional methods. Overall, the control group seemed to get their understanding from the parameters, while the experimental subjects got the intuition from the returned string size, which was the original string's length times `count`.

```java
public static String xxx(String string, int count) {
  checkNotNull(string); // eager for GWT.

  if (count <= 1) {
    checkArgument(count >= 0, "invalid count: %s", count);
    return (count == 0) ? "" : string;
  }

  // IF YOU MODIFY THE CODE HERE, you must update xxxBenchmark
  final int len = string.length();
  final long longSize = (long) len * (long) count;
  final int size = (int) longSize;
  if (size != longSize) {
    throw new ArrayIndexOutOfBoundsException(
    "Required array size too large: " + longSize);
  }

  final char[] array = new char[size];
  string.getChars(0, len, array, 0);
  int n;
  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);
}
```

Figure 4.9: `repeat` method (version 1).

One experimental treatment session was discard from the analysis due to interference during the session, so we were left with three experimental and three control results.

The control group reached a correct answer in 3.5–6.5 minute, while the experimental group needed 5–11 minutes. In this case, there was no significant separation between the time it took the control and the experiment treatments to reach a correct answer (Figure 4.10). We attribute this to the experimental group identifying a familiar pattern which gave them a good enough intuition, the output length mentioned above. However, they were a bit hesitant with their answer, starting with "in my opinion" and a confidence level of 60%, and "my guess is" with 80%.

Moreover, participants mentioned that some of the names are confusing.

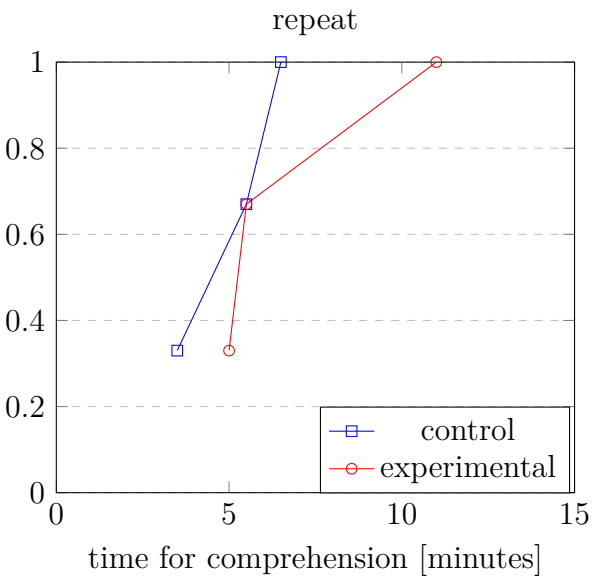

Figure 4.10: Commutative distribution function of required time in the control treatment vs. the experimental treatment.

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.

4.6 abbreviateMiddle

This method abbreviates a String to the given length passed, by replacing its middle with the provided replacement string (Figure 4.11). 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, and to the fact that its variable names are not meaningful enough and even misleading. 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's harder to comprehend the code.

```java
public static String xxx(final String str, final String middle,
    final int length) {
    if (isEmpty(str) || isEmpty(middle)) {
        return str;
    }

    if (length >= str.length() || length < middle.length()+2) {
        return str;
    }

    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();
}
```

Figure 4.11: `abbreviateMiddle` method (version 1). `targetSting` typo in the source.

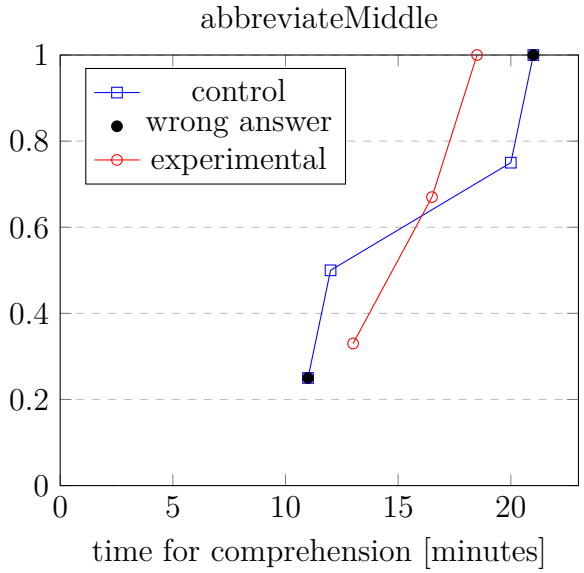

Figure 4.12: Commutative distribution function of required time in the control treatment vs. the experimental treatment.

Table 4.1: Participants suggestions for methods names during the experimental treatment

ID	*Name*	*Suggested name*
1	reverse	iterativeSwap, switchArrayIndexes, swapRange, reverseRange
2	indexOfAny	stringMatcher, firstIndexOfAnySubset, findFirstIndex, indexOfFirstOccurence
3	substringsBetween	getExcludedSubsets, getStringsBetween, splitIntoSubstrings
4	replaceChars	replaceInitialAccur, stringCharReplacer, replace, replaceWith
5	repeat	repeatString, duplicate, multiply, multiplyString
6	abbriviateMiddle	insertSubstringToString, putStringInTheMiddle, replaceMiddleOfString

Perhaps the most offending variable is the int `targetString` (which also had a typo). 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 4.12).

Chapter 5

Discussion

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

5.1 The importance of meaningful names

The results for the 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 [4, 17, 26].

The flip side of meaningful names is misleading names. Two of our methods, `replaceChars` and `abbreviateMiddle`, provide striking examples. With these methods the participants who received the experimental treatment were found to perform better despite the missing identifier names! Moreover, participants in the control treatment, which 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 addition, two methods, `replaceChars` and `indexOfAny`, 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 helps explain why results reported in the literature may be inconsistent with each other.

On a more general note, we claim that limiting coding standards to simple automatically enforceable rules is just not good enough. Our results show the dangers of using such an approach, and emphasize the complexity of choosing good names that will be clear to all or even most developers that would need to access the code in the future. We suggest that naming has such an impact on comprehension that measuring code comprehension quality using simple enforceable rules such as counting the number of lines in a function or other complexity metrics, without giving some kind of weight to its variable naming quality, simply has no value in the evaluation process. For instance, lets examine the following two methods, indexOfAny with 27 lines of code and abbrivateMiddle with 20 lines. While the first took the control treatment 2 to 7 minutes to understand the method functionality, the other took twice the time with two out of the four developers giving the wrong answer regarding its functionality, even thought it was shorter by 7 lines of code.

5.2 Parameters vs. Locals

Summing up which type of identifiers contributes more to the comprehension process, there was a clear advantage for parameters: they were deemed more beneficial by the participants in 79% of the cases.

Figure 5.1 shows the distribution of answers across the different methods. In some cases, notably `reverse`, `indexOfAny`, and `abbreviateMiddle`, pa-

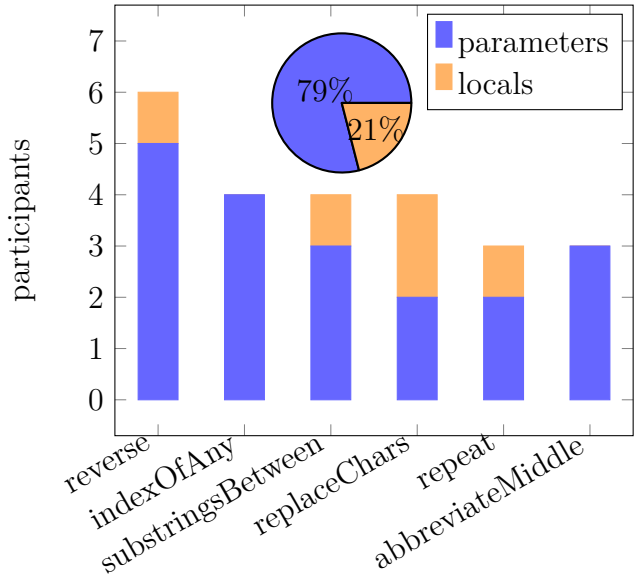

Figure 5.1: parameters vs. locals: which was claimed to be more significant across the different methods.

rameters were nearly universally preferred. This is notable as it implies that when local variables alone were revealed this was not enough, and the participants really needed the parameters in order to understand these methods. We believe this can have two reasons:

- Parameter names were more carefully chosen by the original developers due to being part of the API. In other words, parameters may tend to have comprehension beacons embedded in them because they are part of the interface that defines the method.
- Due to their location in the header, parameters are naturally of higher priority for comprehension. As one subject phrased it: "Parameters are more beneficial, since once I have a clear starting point everything else is easier".

However, in some cases it was not the parameters that mattered, but whatever identifiers were revealed first. This was especially prominent in `substringsBetween`, where the first variables to be revealed — either parameters or locals — were always sufficient for comprehension, or in `replaceChars`

were all subjects stated the first variables which were revealed as more significant to their comprehension process.

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.

Chapter 6

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 [8]. 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 will contain some of the parameter name. For instance, given a parameter called `string`, a method may include a local variable called `stringLength` that holds 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 names.

Another risk is that the attribution of effect to parameters or locals 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.

Chapter 7

Inspirations for Future work

Throughout the process of writing this thesis, many different ideas and directions of research have been discussed. These ideas can be the basis for future related work. In this section we shell suggest some of those directions for future research possibilities.

As mentioned in the background, coding conventions supply some guidelines on naming, though very superficial ones. We propose using industry "state of the art" coding conventions, such as clean code [18], to further examine the hypothesis that there is a lack of good naming guidelines in coding conventions. One can perform the following experiment. Use the same methods used in this study, ask one group of developers to re-factor the methods' variable names according to the guidelines, and perform the control treatment on two other groups to validate if indeed there is a significant difference between the time to comprehension in the original code and the re-factored one. Thus, showing these guidelines benefit the comprehension process of those methods or not, which can further emphasize that naming standards are an important factor that is not given enough attention.

Comparing time to comprehension of different methods in the results of this study suggests that the more meaningless identifiers a method has the longer the comprehension process. Having said that, this suggestion does not apply to misleading identifier names. Is there a direct connection between the numbers of variables and the time to comprehension? Subjects in the experimental treatment have mentioned that having to deal with a larger

number of variables was a memory intensive task. We suggest using studies in human memory research to better understand how many variables can an average human accommodate during the comprehension process. This could result in a good code comprehension metric, one that would better define the upper limit of the number of distinct identifier names allowed in a single function.

One significant result in our study is the higher importance that parameter names have over local variables. Modular code leads to less local variable names and more method and parameter names. Considering that parameters are given better names this could lead to faster comprehension, and give yet another reason why one should write modular code. But proving this require some creative thinking in the experiment design. Moreover, it could be claimed that fragmenting the code up until no local variables are left would result in a faster comprehension process. So, one such experiment could be to re-factor each method up to the stage where no local variable is needed. Then, compare the time for comprehension in the control treatment between the original code and the modular one, which has only method and parameter names.

In the process of choosing our methods we reviewed many GitHub repositories and noticed that there is a wide variety of coding guidelines. Some of which do not contain any guidelines regarding naming. They require consistency more then convention. Others have mostly formatting guidelines. Is code that is contributed to a repository without clear guidelines more or less comprehensible then one with detailed guidelines?

More can be studied about the subjective nature of naming by adding a renaming stage into the experiment. In this stage a group of developers would rename all variable names (and only them) to the best of their ability in order to make the code as clear as possible. Later, two groups of different developers would be given the original code and the renamed one. Comparing time to comprehension between the two groups could suggest that good names are only in the eye of the beholder.

We have concluded that parameters are more important to the comprehension process as part of a larger API. Is this impacted by their public

visibility? Will we get different results dealing with private methods then public ones? To answer this question, one can duplicate our experiments with private methods.

During our experiments, subjects have stated that having two variables with the same meaning is confusing. More specifically, in the repeat method, it was claimed that `size` and `len` have practically the same meaning. Avidan & Feitelson have recently suggested a consistent identifiers metric [2], which tries to estimate the comprehensibility of code by examining the amount of synonyms and homonyms it possesses. Still, there is no implementation of this fairly easy to enforce metric, nor has there been an effort in evaluating how common this phenomenon is. If we were lucky enough to encounter this reviewing only six production code methods, it would be reasonable to believe this is common enough. Given the research friendly API of GitHub, one can conduce a short and effective evaluation of the commonality of the phenomenon, giving motivation to enforce it.

We strongly believe that future research is needed in this field of code semantics. Semantics are one part of what makes a working code a great code, as "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." [11]

Chapter 8

Additional Efforts - Survey on Method Headers

While conducting our experiments, we have started working on a short survey in order to understand the importance of the method header. That is, the method name with its accompanied parameters. If our results are valid and parameter names do hold more information for the comprehension process, then one should better examine how they are named and identify potential issues. For that purpose, we have published a short online questioner with 36 method headers, asking developers to try and explain what is the purpose of each method without seeing the documentation or the code. We had relatively small response rate and have summarized some initial results gathered from 11 male developers. The survey was directed at the same population as our main experiment, the age of participants ranged from 25 to 45, and work experience from 0 to over 15 years (Figure 8.1).

30 of the methods used in the survey were extracted from the `ArrayUtil` class in the Apache commons utility package [1]. The additional 6 were the methods we have conducted our main experiment on. To understand the true functionality of the methods we used either the java docs or method comment. As we did not require an answer for any of the questions, not all methods functionality were supplied by all participants. We continue to describe our initial results.

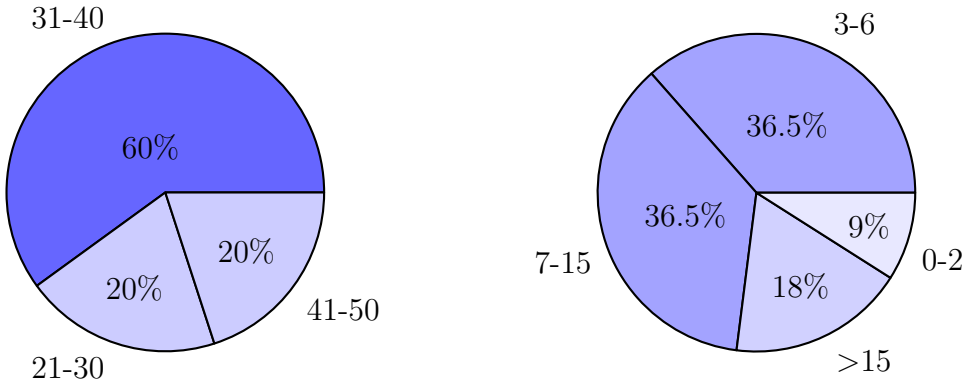

Figure 8.1: participants age range (left) and experience in years (right)

8.1 Initial results

Following are interesting observations of developers and their understanding of method and parameter names in the processes of comprehending the overall functionality of some of the methods. The remaining methods have either displayed similar patterns or had adequate names and were unanimously described correctly.

int indexOfAny(CharSequence str, CharSequence... searchStrs)

As noticed in our experiment, here as well, 4 out of 9 subjects mistreated the var args CharSequence as an array of chars rather then an array of CharSequences. This shows the type notation has a major role in the comprehension process.

float[] addAll(float[] array1, float[]... array2)

The majority of participants, 5 out of 8, stated that this was "a concatenation of two arrays", ignoring the '...' var args in the type annotation of `array2`. This can be attributed to the names `array1` and `array2`, which suggests that they are of the same magnitude. For instance, a better name for array2 could have been arrayList or arrays.

boolean isEquals(Object array1, Object array2)

Only 1 out of 8 subjects described the correct functionality of pointer equality, 2 stated it verifies that all elements in the arrays are equal, and another 5 simply avoided the dilemma and suggested it checks if `array1` is equal to `array2`. We believe this stems from the abstract names given to the parameters and method name. This directs our attention to the importance of giving concise enough names as suggested by [7].

char[] nullToEmpty(char[] array)

Again due to lack of conciseness, only 2 out of 7 subjects stated that it turns a null pointer to an empty array, while the other 5 concluded it replaces all null entries with empty ones in a given array.

char[] removeElement(char[] array, char element)

4 out of 7 developers suggested it "removes the element appearance from the array" without mentioning if it removes only one or all occurrences. 2 developers did get the correct answer stating that the first occurrence is the one removed, but this seems like an educated guess, as there is no indication of this fact in neither the method nor the parameter names.

Object copyArrayGrow1(Object array, Class<?> newArrayComponent-Type)

while 5 out of 8 understood the correct functionality, the other 3 did not give any answer. It seems that the lack of information in the method name and parameters was to high, which led them to simply give up on the method entirely, stating things such as "don't know" without supplying any sort of guess.

String[] substringsBetween(String str, String open, String close)

8 out of 11 answers were correct and the phrasing of the answers suggests that the plural of `substrings` in the method name and the parameters names

`open` and `close` were informative enough. This further strengthen our results that these names were very beneficial to the comprehension process, as stated and seen in the results of our main experiments.

String replaceChars(String str, String searchChars, String replaceChars)

As seen in the control treatment of our main experiment, most (8 out of 9) developers concluded this method conducts a simple replace `substring` in the `str` string. We again claim this is probably due to the type given to the search and replace chars. While the names suggest we are dealing with characters, the type is `String` and is treated as such.

boolean isSameType(Object array1, Object array2)

Only 4 out of 8 were correct, stating that the method "Checks whether two arrays are the same type" Interestingly, 2 participants suggested the method verifies if all elements in the array are of the same type. We believe this could stem from the parameter names which suggest that the functionality is dedicated to arrays rather then plain objects.

char[] removeAll(char[] array, int... indices)

All developers understood the functionality correctly. Interestingly, here everyone noticed that "int ... indices" is a list of things, while this was not the case in previous var args cases. We attribute this to the method name being more concise, and the parameter names "indices" being in plural. This was enough to indicate that the method removes more then one thing, and receives the list of things to remove.

8.2 Discussion

Initial results point, once again, to the subjective nature of naming. As seen in the results, there were two recurring issues: conciseness and name-type inconsistency. When an identifier name (method name or parameter) is not concise enough, answers are either not concise as well or simply wrong. When

there is an inconsistency between a parameter name and its type, were the name and type do not correspond, it leads to ambiguity and confusion.

For instance, the method name `removeElement` is not concise enough. Does it remove the first occurrence of the element, its last or maybe all occurrences? In this case the functionality is to remove the first element. Therefore, we suggest adding it to either the method or parameter names. `removeFirstElement(...)` would be a better choice for this method header.

When it comes to name-type consistency, a common issue we have noticed was the mismatch of plural parameter type with a singular identifier name. When one uses the var args in the type annotation, the name of the corresponding parameter should be in plural. This would prevent confusion. For instance, in `addAll(float[] array1, float[]... array2)` the name `array2` may suggest a single array when in fact the var args in the type suggest that there are multiple arrays. A simple solution would have been to change the name from `array2` to `arrays`.

These seem like reasonable suggestions, and quite easy to follow. Why then aren't they mentioned in any coding guidelines or enforced in reviews? We believe the avoidance from semantics is done simply since current coding conversion enforcement is done by simple automatic rules. Semantics do not seem like an important enough cause to change them. We have shown that semantics play a major role and make a big difference in the comprehension process. We believe that code reviews and re-factoring can and should focus more on the semantics of identifier names. Moreover, in this day and age, where science is having major improvements in AI, predictive models can be constructed to detect ambiguity in code, and point the developers attention for a proper fix.

8.3 Conclusions

We had a limited amount of participants in our survey. Even so, reviewing these initial results, we have witnessed two reoccurring obstacles to comprehension: conciseness and name-type inconsistency. We believe these issues should be given enough attention when naming identifiers, and propose

adding guidelines followed by short examples, similar to the ones presented in the discussion section, to coding conventions. Those guidelines, though not easily enforceable by automatic rules, could be identified by code review discussions or more advanced predictive models. These are only two examples, and we believe much more semantics guidelines could be identified and formalized. Simple modifications in names can make a big difference, save time in maintenance tasks and prevent future bugs.

Chapter 9

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 to the comprehension process than local variable names. This can be because parameter names are more carefully chosen due to their part in the API, and their role in the definition of the method specification in general. 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. In fact, they demonstrated that misleading names are worse than meaningless names like consecutive letters of the alphabet, and can lead to errors in comprehension. 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.

Possible implications of our work include the following. First and foremost, names must be picked with caution and given careful attention. Each variable name should reflect the concept or role represented by this variable. But different people have different mindsets, so it is important to be aware of the difficulties and to make a special effort to find names that are as clear and indicative of the intended meaning as possible.

More generally, our work highlights the need for additional research on how names are interpreted and how better names can be chosen. It would

also be interesting to map additional examples of misleading names, and try to identify common traits that make them bad.

There are also recommendations for practitioners. For example, code reviews may benefit from special attention to the review of names, making sure that all participants interpret the names in the same way. A very ambitious goal would be to devise tools capable of effective automatic evaluation of meaningful names. Due to the subjective nature of naming, this currently seems to be far beyond our reach.

Bibliography

[1] Apache, "*ArrayUtils*". 2016.
URL `https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/ArrayUtils.html`

[2] E. Avidan and D. G. Feitelson, "*From obfuscation to comprehension*". In *2015 IEEE 23rd International Conference on Program Comprehension*, pp. 178–181, IEEE, 2015.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] F. Deissenboeck and M. Pizka, "*Concise and consistent naming*". *Software Quality Journal* **14(3)**, pp. 261–282, 2006.

[8] 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.

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

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

[11] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[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] Google, "*Google Java Style Guide*". 2016.
URL `https://google.github.io/styleguide/javaguide.html/`

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] V. Raychev, M. Vechev, and A. Krause, "*Predicting program properties from "big code"*". In 42nd *Ann. Symp. Principles of Programming Languages*, pp. 111–124, Jan 2015, DOI:10.1145/2775051.2677009.

[25] 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.

[26] 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.

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

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

[29] 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.