2013 IEEE International Conference on Software Maintenance

Dealing with Faults in Source Code: Abbreviated vs. Full-Word Identifier Names

Giuseppe Scanniello
Dipartimento di Matematica, Informatica e Economia
Universitá della Basilicata, Italy
Email: giuseppe.scanniello@unibas.it

Michele Risi
Departimento di Management & Information Technology
Universitá di Salerno, Italy
Email: mrisi@unisa.it

***Abstract*—We carried out a controlled experiment to investigate whether the use of abbreviated identifier names affects the ability of novice software developers to identify and fix faults in source code. The experiment was conducted with 49 students in Computer Science. The results of the statistical analyses indicate that there was not a significant difference to identify and to fix faults, when source code contains either abbreviated and full-word identifier names. In other words, it seems that abbreviated identifiers provide the same information as full-word identifiers on the solution domain and the implementation.**

I. Introduction

Software maintenance, testing, quality assurance, reuse, and integration are examples of activities in the software engineering field that involve existing source code [6]. To deal with these activities software engineers should have source code and design documentation [28]. The worst case scenario is that source code (program statements and source code comments) is the only available resource. However, even if design documentation is available developers consider it inadequate to comprehend source code (e.g., [21], [26], [27]). Therefore, for developers the only possible strategy is the analysis of the source code and/or its execution.

When reading source code, software engineers have two main source of information: identifier names (or simply identifiers, from here on) and comments [22], [23]. When source code is uncommented, as often happens, comprehension is almost exclusively dependent on the identifier names. Three different kind of identifiers can be used: single letters, full-words, and abbreviations. Single letter identifiers are single chars, while full-word identifiers are composed by full English words concatenated according to a given naming convention. Finally, abbreviation variants are based on the full-word variants. Compounding English words are shortened by applying any method. For example, the word *abbreviation* could be abbreviated as *abbr*, *abbrv* or *abb*. A special kind of abbreviation is the contractions. A contraction of a word is made by omitting certain letters and bringing together the first and last letters (e.g., *dr* is the contraction of the word doctor, while *prt* of print).

In this paper, we present the results of a controlled experiment aimed to investigate whether the presence of abbreviated identifiers in source code affects the ability of novice software developers to identify and fix faults. The participants were asked to identify and fix faults in four applications implemented in C. Comments were omitted from the code the subjects viewed. Depending on the application a number of bug reports were given to the participants. The experiment was conducted in a research laboratory at the University of Basilicata with 49 Bachelor students in Computer Science.

The remainder of the paper is organized as follows. In Section II, we present the design of the controlled experiment, while the achieved results are presented and discussed in Section III and in Section IV. We highlight the threats that could affect the validity of our study in Section V. Related work is presented in Section VI, while final remarks conclude the paper.

II. Controlled Experiment

We carried out an ABBA-type experiment [34]. The experiment (denoted E-UBAS) was carried out at the University of Basilicata in January 2012 with 49 students from the Bachelor's program in Computer Science.

The experiment was carried out by following the recommendations provided by Juristo and Moreno [14], Kitchenham *et al.* [19], and Wohlin *et al.* [34] and reported according to the guidelines suggested by Jedlitschka *et al.* [13]. For replication purposes, the experiment material is available at www2.unibas.it/gscanniello/Identifiers/.

A. Goal

Applying the Goal Question Metric (GQM) template [4], the goal of the experiments can be defined as:

- ***Analyse*** abbreviated and full-word identifier names ***for the purpose*** of evaluating their use ***with respect to*** fault detection and fixing ***from the point of view of*** the developer ***in the context of*** C programs and novice software developers.

The use of GQM ensured us that important aspects were defined before the planning and the execution of the experiment took place [34].

B. Context Selection

To reduce external validity threats, we downloaded the used software from the web. The domains of these software can be considered a good compromise of generality and industrial application. Larger and more complex software would be difficult to use in the experiment, so increasing the risks of failure [19]. In particular, we used four software implemented in C. The following are the missions (or problem statements) of these systems:

- **Agenda**[1]. It provides easy to use means for keeping track of personal contacts. It allows adding a new contact by specifying name, last name, telephone number, mobile number, birthday, and email address. Existing contacts can be searched by specifying one of the fields above. All the records stored in the system can be sequentially browsed. The user can load a file containing contacts, so letting to separately manage different contact lists.
- **Financial**[2]. It is a command line option price calculator. It uses Black-Scholes that is a mathematical description of financial markets and derivative investment instruments. The model develops partial differential equations whose solution (i.e., the BlackScholes formula) is widely employed to price European puts and calls on non-dividend paying stocks.
- **GAS-Station**[3]. It is a system to manage a GAS station. It takes as input the daily price of: Petrol-Oil, Diesel, and Compressed Natural GAS. The system is able to manage receipts and bills regarding the refueling of Petrol-Oil, Diesel, and Compressed Natural GAS. The bills are stored to be successively searched, modified, and browsed.
- **Hotel-Reservation**[4]. It is in charge of managing room reservations for an Hotel. To reserve a room the system asks for the fiscal code number of a client and the dates for the check-in and check-out. Reservations can be modified and deleted. The system also shows the status of all the rooms of the hotel.

The source code of the systems above contained some identifiers either abbreviated (e.g., `mobNum`, `lDsk`, or `buf` in Agenda) or not. Acronyms were also used as identifiers (e.g., `CNG` - Compressed Natural Gas - in GAS-Station). Comments were omitted from the code the subjects viewed. One of the authors created two versions for each system chosen: one version contained abbreviated identifiers, while the other full-word identifiers. This process was founded on the dictionary available at www.cs.tut.fi/tlt/stuff/misc/babel.html. In Table I, we report some details on the original versions of the systems chosen. The first column shows the name of the system. The number of Line Of Code (LOCs) is reported in the second column and the total number of identifiers is shown in the third column. The percentage of abbreviated identifiers is reported in the last column.

[1]www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=2299&lngWId=3

[2]www.paulgriffiths.net/program/c/finance.php

[3]www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=11639&lngWId=3

[4]www.vyomworld.com/source/code.asp?id=01&l=C_Projects&t=HotelReservationSystem

Table I
EXPERIMENT OBJECTS

System	LOCs	# Identifiers	Abbrev. Identif. (%)
Agenda	867	44	29.5%
GAS-Station	1021	51	37.2%
Financial	397	37	37.8%
Hotel-Reservation	511	24	50%

Table II
FAULTS INJECTED WITH THE KIMS MUTATION OPERATORS [16], [17]

System	LCO	LOR	CFD	VRO	# faults
Agenda	2	1	1	0	4
GAS-Station	1	1	1	1	4
Financial	0	1	0	1	2
Hotel-Reservation	0	0	1	1	2

We injected the same faults in each variant (with abbreviated identifiers and full-word identifiers) of a given system. The injection process was based on the Kims mutation operators [16], [17]. Some of these operators are general and can be applied to any imperative programming language, while others are specific to object-oriented languages. We used the following subset of the mutation operators for imperative programming languages:

- LCO (Literal Change Operator) - Increase/decrease numeric values or swap Boolean literals.
- LOR (Language Operator Replacement) - Replace a language operator with other legal alternatives.
- CFD (Control Flow Disrupt operator) - Disrupt normal control flow: add/remove break, continue, return.
- VRO (Variable Replacement Operator) - Replaces a variable name with other names of the same type and of the compatible types.

As for Agenda, we injected four faults applying the mutation operators: LCO, LOR, and CFD. For example, we applied the LOR operator on the statement `if(temporaryNode != NULL)`, so obtaining `if(temporaryNode == NULL)`. Details on the performed mutations are summarized in Table II. The application of a mutation operator depends on the source code of a given software. We also tried as much as possible to balance the number of faults within each software: 4 faults in Agenda and GAS-Station (the largest) and 2 in Financial and Hotel-Reservation (the smallest). In addition, we balanced the application number of the mutation operators on the size of the software. The mutated versions of the four software used in the experiment were syntactically correct.

C. Participants

We conducted the experiment under controlled conditions using *convenience sampling* from the population of novice software developers with *students as participants*. The participants had the following characteristics:

- They were students of a course on *Algorithms and Data Structures*. The participants had passed all the exams related to the following courses: Procedural Programming and Object Oriented Programming I. In these courses the participants studied C/C++ and Java on university problems. The participants were 21 years old on average.

The students participated in the experiments on a voluntary basis: we did not force and we did not pay them for their participation. However, we awarded the students for their participation to the experiments with a bonus towards their final mark. Before the experiment, we clearly informed the participants about that.

D. Variable Selection

We considered the source code with extended identifiers as the *Control Group* and the group with abbreviated identifiers as the *Treatment Group*. Therefore, the independent variable was *Method*. It is a nominal variable that could assume the following two values: ABBR (source code with abbreviations) and FULL (source code with full-word identifier names).

The direct dependent variables are:

- *Completion time* - the time which the participant spent to accomplish the experiment task.
- *Identified faults* - the number of faults the participants correctly identified.
- *Fixed faults* - the number of faults the participants correctly fixed in the source code.

To calculate the first dependent variable, we used the time (expressed in minutes) to accomplish the task, which was directly recorded by each participant. Low values for the time mean that the participants were quicker in completing the experiment task.

The variables were measured checking the source code the participants gave back. As for identified faults, we asked the participant to manually mark the statement/s containing the fault by using the following source code comments: `/* fault start */` and `/* fault end*/`. Since we knew the injected faults, the values for the variable identified faults were easily to compute. This variable assumes values between 0 and the maximum number of faults injected (e.g., four for the system Agenda). The larger the value of that variable, the better was the ability of the participant to found faults in the source code.

As for the variable fixed faults, we checked if the participant correctly modified the statement/s affected by the mutation operator. Also in this case the values for that variable were easy to compute because we had the original version of the system as the oracle. Therefore, each fault was correctly identified if and only if the statement the participant wrote was equivalent to that in the oracle. The larger the value of the variable fixed faults, the better was the ability of the participant to remove faults. For scant relevance, we did not consider the number of wrongly identified and fixed faults in source code.

The chosen variables complement each other - one describes the efficiency of the participant and the other two the quality of dealing with faults.

We also analyzed the effect of other independent variables (also called co-factors, from here on):

Object. It denotes the combination system (i.e., Agenda plus Financial and Gas-Station plus Hotel-Reservation) used as the **objects** in the experiment. The effect of the Object factor should not be confounded with the main factor.

Trial. It denotes in which experiment trial a particular participant was exposed to ABBR and FULL. As the participants worked on two different experimental objects in two laboratory trials/runs. We analyzed whether the order might affect the results.

E. Hypotheses Formulation

The following three null hypotheses have been formulated and tested:

Hn0: There is no statistically significant difference between the completion time for ABBR and FULL.

Hn1: There is no statistically significant difference between the number of identified faults with ABBR and FULL.

Hn2: There is no statistically significant difference between the number of fixed faults with ABBR and FULL.

The hypotheses are two-tailed because we did not expect a positive nor a negative effect of the abbreviated identifiers on the experiment tasks. Even though it can be postulated that the participants in the treatment group were provided with less information, we cannot hypothesize that this concern reduces ambiguities and hinders the identification and the removal of faults in the source code.

F. Design of the experiment

We used a within-participants counterbalanced experimental design (see Table III). This design ensures that each participant works on different experiment objects (i.e., A+F = Agenda plus Financial and G+H = GAS-Station plus Hotel-Reservation) in two laboratory trials/runs, using ABBR or FULL each time. We opted for that design because it is particularly suitable for mitigating possible carry-over effects[5]. We grouped the systems in the experiment objects according to their size, the number of injected faults, and the application domain.

[5]If a participant is tested first under the condition X and then under the condition Y, he/she could potentially exhibit better or worse performances under the second condition.

Table III
EXPERIMENT DESIGN

Trial	Group A	Group B	Group C	Group D
First	A+F, ABBR	A+F, FULL	G+H, ABBR	G+H, FULL
Second	G+H, FULL	G+H, ABBR	A+F, FULL	A+F, ABBR

We used the participants' average grades as blocking factor: the groups in Table III are similar to each other with respect to the number of participants with high and low average grades[6]. The groups A and B contained 13 students, while 12 and 11 were into the groups C and D, respectively.

G. Experiment Tasks

We asked the participants to perform the following tasks:

1) *Identifying the faults*. The participants were asked to identify faults in the source code of the two experimental objects according to the design of the experiment. For example, the participants in the group A were asked to work first on the object A+F with ABBR and then on the object G+H with FULL. Regarding the object A+F, we asked the participants to work first on Agenda and then on Financial. The participants were asked to work first on GAS-Station and then on Hotel-Reservation when dealing with the object G+H. For each system, the participants were provided with its mission (or problem statement) describing its intent and a bug report for each fault. The bug reports contain both the title of the bugs (i.e., a short description) and their (long) description. An example of bug report for Agenda is shown in Figure 1. The bug report of a system is the same independently from the method (i.e., ABBR and FULL). Bug reports contained neither abbreviated nor full-word identifier names. It is also worth mentioning that the participants were not provided with any tools to perform fault localization in source code.
2) *Fixing the faults*. The participants were asked to remove the faults previously identified.
3) *Filling in the post-experiment questionnaire*. We asked the participants to fill in a post-experiment survey questionnaire. This questionnaire contained questions about: the availability of sufficient time to complete the tasks and the clarity of the experimental material and objects. The post-experiment survey questionnaire is shown in Table IV. All the statements expected closed answers according to a five-point Likert scale [25]: (1) strongly agree; (2) agree; (3) neutral; (4) disagree; and (5) strongly disagree. The goal of that questionnaire was to obtain feedback about the participants' perceptions of the experiment execution.
4) *Expanding abbreviations*. The participants were also asked to expand all the abbreviated identifiers in the source code they studied in the first two tasks above when using ABBR. The goal was to assess whether or not the participants correctly associated full-word identifiers to the abbreviated ones.

[6]In Italy, the exam grades are expressed as integers and assume values in between 18 and 30. The lowest grade is 18, while the highest is 30. As suggested by Abrahão *et al.* [1], we consider here the average grade as low if it is below 24/30, high otherwise.

Table IV
POST-EXPERIMENT SURVEY QUESTIONNAIRE

Id	Question
Q1	I had enough time to perform the tasks.
Q2	The task objectives were perfectly clear to me.
Q3	The bug reports were clear to me.
Q4	The source code was correctly indented.
Q5	I found difficult to understand the abbreviated identifiers.
Q6	The presence of abbreviated identifiers made difficult the identification and the fixing of faults in source code.
Q7	There is not difference in using abbreviated or full-word identifiers while identifying and fixing faults in source code.
Q8	I found useful the experiment from the practical point of view.

To perform both the laboratory trials, the participants were provided with laptops having the same hardware configuration (i.e., equipped with a 2.93 GHz i3-560 Processor with 4 GB of RAM and Windows 7). To surf, run, execute, and debug source code, we installed on each laptop Dev-C++ (available at sourceforge.net/projects/orwelldevcpp/) an Integrated Development Environment (IDE) for C/C++. We opted for this IDE because it widely used in academic programming courses (e.g., in the procedural programming course). In addition, Dev-C++ can be considered easy to learn and to master.

The identifiers and the bug reports were in English. The complete list of all the bug reports and the source code of the four systems (both with abbreviated and full-word identifiers) are available on the web page of our experiment.

H. Experiment operation

The participants first attended an introductory lesson in which the supervisors presented detailed instructions on the experiment. The supervisors highlighted the goal of the experiment without providing details on the experimental hypotheses. The participants were informed that the data collected in the experiments were used for research purposes and treated confidentially.

Title	Search for telephone number
Description	The search by telephone number does not produce any output. In particular, when a telephone number is provided as input and it is present in the contact list the system shows the message "no matches found".

Figure 1. A sample fault of the Agenda system

To familiarize with the experimental procedure, the participants accomplished an exercise similar to those which would appear in the laboratory trials. The software used in that exercise was the "Tower of Hanoi". We download it from www.paulgriffiths.net/program/c/hanoi.php.

After the introductory lesson and training session, the participants were assigned to the groups A, B, C, and D (see Table III). No interaction was permitted among the participants, both within each laboratory trial and while passing from the first trial to the second one. No time limit for performing each of the two trials was imposed.

To carry out the experiment, the participants first received the material for the first laboratory trial. When the participants had finished, the material for the second trial was provided to them. After the completion of both the trials, we gave the participants the post-experiment questionnaire.

We asked the participants to use the following experimental procedure: *(i)* specifying name and start-time; *(ii)* identifying (and marking) the faults from the bug reports; *(iii)* fixing the bugs; and *(iv)* marking the end-time. The participants could compile and execute the applications before and after each modification performed on the source code. The debugger of Dev-C++ could be also used. We did not suggest any approach to surf, run, execute, and debug source code.

I. Analysis Procedure

We performed the following steps:

1) We calculated the descriptive statistics of the dependent variables.
2) We tested the null hypotheses using unpaired analyses because the experimental tasks were accomplished on two different objects (see Table III). We have planned to use unpaired t-test when the data follow a normal distribution. The normality has been verified using the Shapiro-Wilk W test [30] (simply Shapiro in the following). A p-value smaller than the α threshold allows us to reject the null hypothesis and to conclude that the distribution is not normal. If the data are not normally distributed, our non-parametric alternative to the unpaired t-test was the Wilcoxon rank-sum test (also known as Mann-Whitney) [9].
   The chosen statistical tests analyze the presence of a significant difference between independent groups, but they do not provide any information about that difference [15]. Therefore, in the context of the parametric analyses, we used Cohen's d [8] effect size to obtain the standardized difference between two groups. That difference can be considered: negligible ($|d| < 0.2$), small ($0.2 \leq |d| < 0.5$), medium ($0.5 \leq |d| < 0.8$), and large ($|d| \geq 0.8$) [15]. Conversely, we used the point-biserial correlation r in case of non-parametric analyses. The magnitude of the effect size measured using the point-biserial correlation is: small ($0 < r \leq 0.193$), medium ($0.193 < r \leq 0.456$), and large ($0.456 < r \leq 0.868$) [15].
   We also analyzed the statistical power for each test performed. The statistical power is the probability that a test will reject a null hypothesis when it is actually false. The statistical power is computed as 1 minus the Type II error (i.e., β-value). This kind of error indicates that the null hypothesis is false, but the statistical test erroneously fails to reject it. In the discussion of the results, the β-value has to be used when a statistical test is not able to reject the null hypotheses. The higher the β-value, the less is the likelihood of not rejecting a null hypothesis when it is actually false.
3) To analyze the influence of the co-factors, we planned to use a two-way Analysis of Variance (ANOVA) [10]. To apply this test data have to be normally distributed and their variance has to be constant. The normality and the variance of the data were tested using the tests of Shapiro and Levene [24], respectively. In case these two assumptions are not verified, we would use a two-way permutation test [2], a non-parametric alternative to the two-way ANOVA test.
4) To graphically summarize the answers to the post-experiment survey questionnaire, we planned to use boxplots. They provide a quick visual representation to summarize the data using five numbers: the median, upper and lower quartiles, minimum and maximum values, and outliers.

In all the statistical tests, we decided (as custom) to accept a probability of 5% of committing Type-I-error [34] (i.e., the α threshold is 0.05). The R environment[6] for statistical computing has been used in the data analyses.

III. Results

In this section, we present the data analysis following the procedure presented above.

A. Descriptive statistics and exploratory analysis

Table V reports some descriptive statistics (i.e., median, mean, and standard deviation) of the three dependent variables on Method. A summary of the performed analyses are also shown: the p-values, the effect size, and the statistical power (β-values are between brackets). The observations are grouped by Method.

The descriptive statistics show any tendency in favor of neither ABBR nor FULL. In particular, we can observe that the participants achieved slightly larger values for completion time when using ABBR. As for the other two dependent variables, descriptive statistics are almost the same. In addition, the descriptive statistics suggest that the participants generally were able to fix all the correctly identified faults.

[6]www.r-project.org

Table V
DESCRIPTIVE STATISTICS AND RESULTS. THE SYMBOL "*" INDICATES THAT PARAMETRIC ANALYSES HAVE BEEN PERFORMED

	FULL			ABBR			p-value	effect size	statistical
Var.	median	mean	st. dev.	mean	st. dev.	median		d or r	power
Completion time*	73	74.92	32.006	77	78.02	33.503	0.64	negligible (-0.095)	0.054 ($\beta - value$ = 0.946)
Identified faults	4	4.286	1.307	4	4.204	1.338	0.748	small (0.068)	0.063 ($\beta - value$ = 0.937)
Fixed faults	4	3.837	1.297	4	3.816	1.269	0.93	small (0.149)	0.047 ($\beta - value$ = 0.953)

B. Effect of method

The data were normally distributed on completion time. The Shapiro test returned 0.516 and 0.881 as the p-values for ABBR and FULL, respectively. Then, we applied parametric analyses. In particular, the results suggest that Hn0 (effect of Method on completion time) cannot be rejected since the p-value is 0.64. The $\beta - value$ is 0.946 (i.e., it is high the likelihood that the statistical test correctly fails to reject the null hypothesis). The effect size is negligible (-0.095).

As far as the other two dependent variables, we performed non-parametric analyses because the data were not normally distributed. The Mann-Whitney test failed to reject both the hypotheses Hn1 (effect of Method on identified faults) and Hn2 (effect of Method on fixed faults). The p-values are 0.748 and 0.93, respectively. For both the hypotheses the $\beta - values$ are high: 0.937 and 0.953, respectively. The effect size is always small (i.e., 0.068 and 0.149 on the identified and fixed faults, respectively).

C. Other factors

The results of the analysis of the co-factors is summarized in Table VI. This table reports the p-values of the statistical tests employed to verify whether or not a co-factor has any effect on each of the dependent variables. The results about the interaction between the co-factors and Method are reported as well. We could apply a two-way ANOVA on Object and Trial for completion time. As far as Object, the Shapiro test returned 0.758 and 0.999 as the p-values for A+F and G+H on ABBR, respectively. For FULL, this test returned as the p-values 0.461 for A+F and 0.695 for G+H. The Levene test returned 0.793 as the p-value for Method and 0.695 as the p-value for Object. As for Trial, the p-values obtained by applying the Shapiro test on ABBR are: 0.786 (first trial) and 0.729 (second trial). On the other hand, the p-values are 0.607 and 0.611 for the first and the second laboratory trials, respectively. The Levene test returned 0.229 as the p-value for Trial.

As for the other two dependent variables, the assumptions to use a two-way ANOVA were violated and then we applied a two-way permutation test.

1) Object: The results of a two-way ANOVA show that the effect of Object on completion time was not statistically significant (p-value = 0.318) and that there was not a statistically significant interaction between Method and Object (p-value = 0.688).

Table VI
RESULT OF THE ANALYSIS ON THE CO-FACTORS. THE SYMBOL "*" INDICATES THAT PARAMETRIC ANALYSES HAVE BEEN PERFORMED

Variable	Object	Object vs. Method	Trial	Trial vs. Method
Completion time*	0.318	0.688	**0.002**	0.987
Identified faults	0.941	0.156	0.882	0.902
Fixed faults	0.882	0.119	0.36	0.921

As far as the variable identified defects, the results of the two-way permutation test indicate that the effect of Object was not statistically significant (p-value = 0.941). Also, the interaction between Object and Method was not statistically significant (p-value = 0.156). Similar results were achieved for the variable fixed faults. For Object, the p-value is equal to 0.882. For the interaction between Object and Method, the p-value is 0.119.

2) Trial: The results of a two-way ANOVA show that the effect of Trial on completion time was statistically significant. The p-value is 0.002 (highlighted in bold in Table VI). The descriptive statistics show that the participants spent less time to accomplish the second laboratory trial (medians: 62 vs. 81; mean 66.51 vs. 86.43). The results of the two-way ANOVA test show that there is not a statistically significant interaction between Method and Trial: the p-value is equal to 0.987.

The results of the two-way permutation test show that there was not a statistically significant effect of Trial on the variables identified faults (p-value = 0.882) and fixed faults (p-value = 0.36). The results of that test also indicate that the interaction between Trial and Object is not statistically significant. The p-values are 0.902 and 0.921 for the variables identified faults and fixed faults, respectively.

D. Post-experiment survey questionnaire

Figure 2 shows the boxplots of the answers the participants gave to the statements of the post-experiment survey questionnaire. The participants judged adequate the time to perform the tasks (Q1). The medians is equal to 1 (strongly agree). The participants agreed on the fact that the task objectives were clear (Q2). The median is 2 (agree). For Q3 (bug report clear) and Q4 (source code correctly indented), the median is 2 (agree). The participants found difficult to: *(i)* understand abbreviated identifiers (Q5) and *(ii)* accomplish the tasks when abbreviated identifiers were present in the source code (Q6). The median is 2 (agree) for both Q5 and Q6. The participants also asserted that there

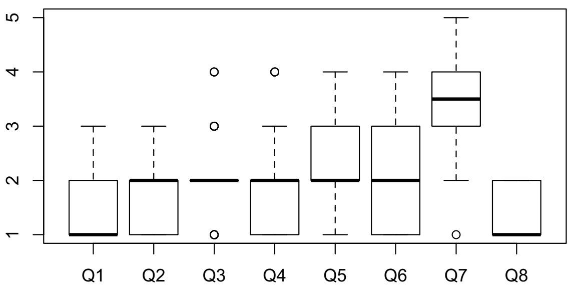

Figure 2. Boxplot of the answers to the survey questionnaire

is a difference in identifying and fixing faults when either abbreviated or full-word identifiers are in the source code (Q7). The median is 4 (disagree). The results on Q5, Q6, and Q7 contrast with the quantitative ones presented in Section III-B. This is the case where there is difference between perceived and effective complexity in the execution of a task. Finally, the answers to Q8 suggested that the participants found useful the experiment from the practical point of view. The median is 1 (strongly agree).

IV. Discussion

Our "null-results" (i.e., the defined null hypotheses have not been rejected) are interesting because they falsify the *common sense* that a difference should be measured when dealing with and without abbreviated identifier name in the execution of comprehension tasks [23]. In practical terms, the two different representations for source code identifiers would transmit the same underlying conceptual content and its mean has been similarly inferred by the participants to our experiment when dealing with faults in source code. That is, abbreviated identifiers did not penalize the identification and the removal of faults in source code. To get some indications to explain the results above, we performed unstructured interviews with the participants after the data analyses. We observed that all the participants daily use social media, such as Facebook, Google+, and Twitter, and are comfortable with the social media slang that they prevalently use (often unconsciously) to communicate. Therefore, it could be possible that the familiarity with the social media slang, which is strongly based on abbreviations, improves the ability to infer the underlying conceptual content of abbreviated identifiers. Although this point deserves future investigations to increase our awareness in the achieved results, we believe that it delineates some research trends in the field of software engineers. The most interesting trend is: *how social media slang changes the programmers' habit in writing and comprehending source code*.

With regard to completion time, the results showed that the participants spent more time to accomplish the assigned tasks when the source code contained abbreviated identifiers. The difference in the execution of the tasks when identifiers are abbreviated or not is not statistically significant (see Table V). We can postulate that this difference is due to the additional time the participants needed to infer the underlying conceptual content of the abbreviated identifiers. This postulation is also corroborated by the fact that the effect of Object is not statistically significant.

We observed that the effect of Trial is statistically significant on completion time, while it is not on the other two dependent variables. This result suggests that an increasing familiarity of the participants with the kind of task and/or with the used IDE affected the time to complete the second trial, but not the ability of these participants to identify and fix faults in source code.

A. Implications

To judge the implications of our experiment, we adopted a perspective-based approach [3]. In particular, we based our discussion on the *practitioner/consultant* (simply *practitioner* in the following) and *researcher* perspectives [18]. The main practical implications of our study can be summarized as follows:

- The ability of participants to identify and fix faults in source code seems independent from how the identifiers are written. From the practitioner perspective, this result is relevant because source code could be written without paying attention on the names of identifiers. From the researcher perspective, it is interesting to investigate whether variations in the context (e.g., larger systems and more or less experienced developers) lead to different results. The researcher could be also interested in studying why the participants perceived more complex the execution of the tasks on source code with abbreviated identifiers (see Section III-D).
- The time to infer the underlying conceptual content of abbreviated identifiers is slightly larger than that of expanded identifiers. This result is relevant for the researcher because it is interesting to investigate how participants (independently from the experience) associate the underlying conceptual content to the identifiers. In addition, the researcher could be interested in analyzing whether writing source code with contractions/abbreviations reduces/increases the effort and cost to software development. On the other hand, the practitioner could be interested in the relation between the effort and cost to write source code with and without abbreviated identifiers and the effort and cost to identify and fix faults.
- The study is focused on desktop applications implemented in C. The researcher and the practitioner could be interested in answering the question: do the results observed hold for different kinds of software (e.g., web application) and/or implemented in different programming languages (e.g., PHP and Java)?

- The software used in the experiment was realistic enough. Although we are not sure that our findings scale to real systems, the obtained results could be true in all the cases in which fault detection is executed on small/medium sized software. Both practitioners and researchers could be interested in this concern.
- The achieved results also suggest that the task completion time and fault identification and fixing are not directly related. Our study poses the bases of future investigations in that direction. This result is interesting from the researcher and practitioner perspectives.

V. Threats to Validity

In the following subsections, we discuss the possible threats that could affect the validity of our experiment.

A. Internal validity

Internal validity threats are diminished by the design of the experiments we adopted. Each group of participants worked on two different experiment objects, each containing either abbreviated or full-word identifiers.

Internal validity threat was also mitigated because the participants had a similar amount of experience with the C programming language, computer programming, and software maintenance. Furthermore, all the participants found the material, the tasks, and the goals clear, as the post-experiment survey questionnaire results showed.

Another issue concerns the exchange of information among the participants. The participants were not allowed to communicate with each other. We prevented this by monitoring them both during the laboratory trials and during the break between the two trials.

B. External validity

The use of students could lead doubts concerning their representativeness with regard to software professionals. The tasks to be performed did not require a high level of industrial experience, so we believed that the use of students could be considered appropriate, as suggested in the literature [7] [12]. In addition, students could be more comfortable than senior software practitioners with contractions/abbreviations since they daily use social networks. This implies that the participants to our experiment could be more proficient in inferring the right meaning to abbreviated identifier with respect to senior professionals.

Working with students also implies various advantages, such as the fact that the students' prior knowledge is rather homogeneous, there is the possible availability of a large number of participants [33], and there is the chance to test experimental design and initial hypotheses [31]. An additional advantage in using students is that the cognitive complexity of the experiment objects is not hidden by the experience of the participants.

Another threat to external validity concerns the used experiment objects. The experimenters were not involved in the realization of these objects, so reducing such a possible threat. The size of the used systems could also threaten external validity as well as the language used for the identifiers. Given the context of the experiment, the familiarity of the participants with the English language could have affected the observed results. In addition, the mutation operators applied on the original versions of the chosen software systems could affect external validity, namely different faults could lead to different results.

C. Construct validity

This kind of validity may be influenced by the used measures, the post-experiment survey questionnaire, and social threats. We used a well-known method [16], [17] to inject faults in a software and used its original version as the oracle to compute the number of identified and fixed faults. In addition, the injected faults were sufficiently complex without being too obvious. We also tried as much as possible to inject different kinds of faults. The post-experiment survey questionnaire was designed using standard forms and scales [25]. Finally, the students were not graded on the results obtained in the experiments to avoid social threats (i.e., evaluation apprehension).

D. Conclusion validity

The used statistical tests to reject the null hypotheses could affect conclusion validity. Depending on the cases, we used unpaired parametric and non-parametric tests. A power analysis has been performed as well. Threats to the conclusion validity could be also concerned to our unstructured interviews.

VI. Related Work

Source code comprehensibility and modifiability have been largely investigated in software maintenance field through quantitative and qualitative studies (e.g., [11], [27], [29], [35]). For example, Woodfield *et al.* [35] reported on an experiment to investigate how different types of modularization and comments are related to programmers' ability to understand programs. The experimenters used eight versions of the same program. These versions were the result of the program being constructed with four types of modularization (i.e., monolithic, functional, super, and abstract data type) each with and without source code comments. The participants to this study were 48 experienced programmers. The comprehension level achieved by these participants was measured by using a questionnaire. The results indicated that those participants whose programs contained comments were able to answer more questions than those without comments. Similarly, Takang *et al.* [32] conducted an controlled experiment founded on program comprehension theories.

Three were the investigated hypotheses: *(i)* commented programs are more understandable; *(ii)* programs that contain full-word identifier names are more understandable; and *(iii)* the combined effect of comments and identifier names tend to enhance the understandability. The experiment was conducted on four versions of a program written in Modula-2 on two methods. The only hypothesis the authors were able to positively answer was the first. In addition, the authors also observed that the quality of identifier names is an issue that merits closer consideration and exploration in its own right. Based on this finding, our study was concerned with the use of different kinds of identifier names on fault identification and fixing.

Lawrie *et al.* [22] conducted a quantitative study with 100 programmers to investigate whether source code comprehension is affect by how identifier names are written. In particular, the authors considered identifiers of 12 functions (i.e., algorithms studied in computer science courses and functions extracted from production code) written as: single letters, abbreviations, and full-words. The comprehension achieved by the participants on these functions was assessed through comprehension questionnaires. The results showed that full-word identifiers lead to the best comprehension with respect to abbreviated identifiers, even if this difference is not statistically different. Successively, Lawrie *et al.* extended the data analysis [23]. The achieved results somewhat contrast with ours: we did not find a huge difference between the use of full-word and abbreviated identifiers. This difference in the observed results could be related to the familiarity of the participants with the social media slang. In fact, in the 2006 and 2007 (the publication years of [22], [23], respectively) social networks were not yet a part of everyday life. That is, social networks were not yet the standard communication media for young people.

Binkley *et al.* [5] presented a family of studies to investigate the impact of two identifier styles (i.e., *camel case* and *underscore*) on the time and accuracy of comprehending source code. The studies involve 150 participants with varied demographics from two different universities. The results suggested that experienced software developers appeared to be less affected by identifier style, while beginners benefited from the use of camel casing with respect to task completion time and accuracy. Several are the differences between the studies by Binkley *et al.* and that reported in this paper.

As far as qualitative studies, Roehm *et al.* [27] conducted an observational study with professional developers on the strategies they followed, the information needed, and tools used to deal with comprehension tasks. The results of that qualitative investigation showed that developers did not use tools for program comprehension and were unaware of them. Therefore, running the system, source code, and communication among developers represented the only source of information to perform comprehension tasks. This is why we provided participants with the only source code.

Ko *et al.* [20] performed a study to understand how developers gain source code comprehension and how an IDE (i.e., Eclipse) might be involved. The authors found that developers interleaved three activities: searching for relevant code both manually and using search tools; following incoming and outgoing dependencies of relevant code; and collecting code and other information. This was the reason because we decided to provide the participants in the experiment with a software tool to surf, run, execute, and debug source code.

VII. Conclusion

In this paper, we present a controlled experiment to assess whether identifier names have any effect on the identification and the removal of faults in C source code. The results indicated that the difference in using abbreviated and full-word identifiers is not statistically significant with respect to: the time to complete the tasks and the number of faults the participants identified and fixed.

As future work, we are going to study the relation between the ability of the participants in correctly associating full-word identifiers to the abbreviated ones and the number of faults they correctly identified and fixed. It is subject of future research also the investigation on the relation between the familiarity of software engineers with social slang and their comprehension of source code.

Acknowledgment

Special thanks to all the participants as this work would not be possible without your time. Our thanks to the Massimiliano Di Penta for his comments and suggestions.

References

[1] S. M. Abrahão, C. Gravino, E. I. Pelozo, G. Scanniello, and G. Tortora. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Trans. on Soft. Eng.*, 39(3), 2013.

[2] R. Baker. Modern permutation test software. *In E. Edgington, editor, Randomization Tests, Marcel Decker*, 1995.

[3] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, L. S. Sørumgård, and M. V. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, 1(2):133–164, 1996.

[4] V. R. Basili and H. D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Software Eng.*, 14(6):758–773, 1988.

[5] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[6] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In *Workshop on the Future of Software Engineering*, pages 326–341, 2007.

[7] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *Proc. of International Symposium on Software Metrics*, pages 239–. IEEE CS Press, 2003.

[8] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1988.

[9] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd Edition, 1998.

[10] J. L. Devore and N. Farnum. *Applied Statistics for Engineers and Scientists*. Duxbury, 1999.

[11] C. Gravino, M. Risi, G. Scanniello, and G. Tortora. Do professional developers benefit from design pattern documentation? a replication in the context of source code comprehension. In *Proc. of International Conference on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 185–201. Spinger, 2012.

[12] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects: comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, Nov. 2000.

[13] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting experiments in software engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer London, 2008.

[14] N. Juristo and A. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, Englewood Cliffs, NJ, 2001.

[15] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information & Software Technology*, 49(11-12):1073–1086, 2007.

[16] S. Kim, J. A. Clark, and J. A. McDermid. The rigorous generation of Java mutation operators using HAZOP. In *Proceedings of International Conference on Software & Systems Engineering and their Applications*, pages 9–10, 1999.

[17] S. Kim, J. A. Clark, and J. A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proc. of NET.OBJECTDAYS*, pages 9–12, 2000.

[18] B. Kitchenham, H. Al-Khilidar, M. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering*, 13(1):97–121, 2008.

[19] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. on Soft. Eng.*, 28(8):721–734, 2002.

[20] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, Dec. 2006.

[21] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of International Conference on Software engineering*, pages 492–501. ACM, 2006.

[22] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Proc. of International Conference on Program Comprehension*, pages 3–12. IEEE CS Press, 2006.

[23] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[24] H. Levene. Robust tests for equality of variances. In I. Olkin, editor, *Contributions to probability and statistics*. Stanford Univ. Press., Palo Alto, CA, 1960.

[25] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.

[26] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, December 2004.

[27] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of International Conference on Software Engineering*, pages 255–265. IEEE CS Press, 2012.

[28] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico. Using the Kleinberg algorithm and Vector Space Model for software system clustering. In *Proc. of International Conference on Program Comprehension*, pages 180–189. IEEE Computer Society, 2010.

[29] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, and G. Tortora. On the impact of UML analysis models on source code comprehensibility and modifiability. *ACM Trans. on Soft. Eng. and Meth.*, (to appear).

[30] S. Shapiro and M. Wilk. An analysis of variance test for normality. *Biometrika*, 52(3-4):591–611, 1965.

[31] D. I. K. Sjoberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.

[32] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: An experimental study. *Journal of Programming Languages*, 4(3):143–167, 1996.

[33] J. Verelst. The influence of the level of abstraction on the evolvability of conceptual models of information systems. In *Proc. of the International Symposium on Empirical Software Engineering*, pages 17–26. IEEE CS Press, 2004.

[34] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.

[35] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proc. of the international conference on Software engineering*, pages 215–223. IEEE CS Press, 1981.