

Fixing Faults in C and Java Source Code: Abbreviated vs. Full-Word Identifier Names

GIUSEPPE SCANNIELLO, University of Basilicata

MICHELE RISI, University of Salerno

PORFIRIO TRAMONTANA, University of Naples “Federico II”

SIMONE ROMANO, University of Basilicata

We carried out a family of controlled experiments to investigate whether the use of abbreviated identifier names, with respect to full-word identifier names, affects fault fixing in C and Java source code. This family consists of an original (or baseline) controlled experiment and three replications. We involved 100 participants with different backgrounds and experiences in total. Overall results suggested that there is no difference in terms of effort, effectiveness, and efficiency to fix faults, when source code contains either only abbreviated or only full-word identifier names. We also conducted a qualitative study to understand the values, beliefs, and assumptions that inform and shape fault fixing when identifier names are either abbreviated or full-word. We involved in this qualitative study six professional developers with 1–3 years of work experience. A number of insights emerged from this qualitative study and can be considered a useful complement to the quantitative results from our family of experiments. One of the most interesting insights is that developers, when working on source code with abbreviated identifier names, adopt a more methodical approach to identify and fix faults by extending their focus point and only in a few cases do they expand abbreviated identifiers.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**;

Additional Key Words and Phrases: Controlled experiments, ethnography study, family of experiments, qualitative investigation, maintenance, replications, source code identifiers, software testing

ACM Reference format:

Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. 2017. Fixing Faults in C and Java Source Code: Abbreviated vs. Full-Word Identifier Names. *ACM Trans. Softw. Eng. Methodol.* 26, 2, Article 6 (July 2017), 43 pages.

<https://doi.org/10.1145/3104029>

1 INTRODUCTION

Developers should be provided with source code and software documentation to deal with software maintenance, testing, quality assurance, reuse, and software integration activities [12]. A key aspect of all these activities is software comprehension. In this respect, it has been observed that

Authors' addresses: G. Scanniello, DiMIE, University of Basilicata, Viale Dell'Ateneo 10, Potenza, PZ, 85100, Italy; email: giuseppe.scanniello@unibas.it; M. Risi, DI, University of Salerno, Via Giovanni Paolo II 132, Fisciano, SA, 84084, Italy; email: mrisi@unisa.it; P. Tramontana, DIETI, University of Naples “Federico II”, Via Claudio 21, Naples, NA, 80125, Italy; email: ptramont@unina.it; S. Romano, DiMIE, University of Basilicata, Viale Dell'Ateneo 10, Potenza, PZ, 85100, Italy; email: simone.romano@unibas.it.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1049-331X/2017/07-ART6 \$15.00

<https://doi.org/10.1145/3104029>

developers believe inadequate available documentation and tool support (e.g., [41, 54]). This implies that developers very often only focus on the source code of the software under study [56]. It could also happen that developers do not have software documentation and have at their disposal only source code statements and comments [42, 43]. Therefore, developers have three main sources of information to perform software comprehension tasks: source code statements, comments, and the syntactical structure of source code [42, 43]. Also, source code layout could affect its comprehensibility [72]. When source code is uncommented or comments are not up to date, developers might take advantage from source code statements, such as identifier names (or also simply identifiers, from here on), keywords, operators, and literals, and clearly the syntactical structure and the layout of source code.

As for identifiers, we can distinguish three different ways to write their names: single letters, full-word, and abbreviated. Single letter identifiers are composed by a single character, while full-word ones are composed by full words concatenated according to a given naming convention (e.g., *camelCase*). Abbreviated identifiers are full-word variants, where compounding words are shortened by using a given method. For example, the word *abbreviation* could be abbreviated as *abbr*, *abbrv*, or *abb*. An abbreviation might be also made by omitting certain characters and bringing together the first and last ones (e.g., *dr* is a contraction of the word *doctor*, while *prt* is a contraction for the word *print*). Acronyms (e.g., *URL*) and in some cases single letter identifiers (e.g., *i* as *iteration*) can be considered a variant of abbreviated identifier names.

Although there are a number of approaches and empirical studies on source code identifiers (e.g., [3, 21, 27]), only a few evaluations in the context of program comprehension have been conducted so far [11, 42, 43]. This lack is even more evident in software testing, where the effect of identifiers on fault identification and fixing has been marginally investigated (e.g., [64]).

In this article, we present the results of a family of four controlled experiments carried out to investigate whether the fixing of faults in source code is affected if that code contains either abbreviated or full-word identifier names. We focused on three constructs concerned with fault fixing: effort,¹ effectiveness,² and efficiency.³ The original experiment (or baseline) was conducted with 49 Bachelor students in Computer Science at the University of Basilicata [64]. We will refer to this experiment as UniBas, from here on. To further investigate the results obtained in UniBas, we carried out three replications whose participants had different backgrounds and experience. These replications were conducted with Computer Science Master's degree students from the University of Naples (UniNa, from here on), and Software Engineering Master's degree students from the University of Naples (PoliNa). Graduate students took part in the fourth experiment (Prof). They were novice employees of different software companies enrolled in a professionalization program. In UniBas and UniNa, we used the same source code written in the C programming language. In the latter two replications, we asked participants to fix faults in source code written in Java. We introduced this variation to increase our confidence in the results attained in the original experiment first and then in the replication UniNa. To interpret quantitative results, a qualitative study was conducted with six professional developers with 1 to 3 years work experience. In this further study, we sought to understand the values, beliefs, and assumptions of developers asked to fix faults in source code with either abbreviated or full-word identifiers. Given this motivation, our methodological approach could be characterized as ethnographic.

This article is organized as follows. In Section 2, we discuss motivations, background, and related work. We present the design of our family of experiments and obtained results in Section 3 and Section 4, respectively. In Section 5, we show the design of our qualitative study, while its outcomes

¹It is the time to fix faults in source code.

²It is the correctness and completeness to perform fault fixing tasks.

³It is the ability to effectively fix faults in source code without wasting time.

are discussed in Section 6. We discuss overall results in Section 7. This article concludes with our final remarks.

2 MOTIVATION AND BACKGROUND

2.1 Motivation

Software engineering is not the first discipline that studies the effects of using different ways for representing information on human performance [2, 63]. Therefore, we should bring the insights of other research areas to our evaluations. A branch of the cognitive science that treats humans and the artifacts they use to solve problems as a single cognitive entity is known as external cognition. According to Scaife and Rogers [61], there are several ways to represent information that lets us improve reasoning, namely, computational offloading, re-representation, and graphical constraining. For example, re-representation refers to how different representations with the same abstract structure make problem-solving either easier or more difficult. Zhang and Norman [80] designed a study so that in certain conditions participants had to internalize several rules to carry out a task, while in others the same rules were embedded. For example, in the case of multiplication tasks using roman or arabic numerals (both represent the same formal structure), the authors observed that the former is much harder for people used to working with the decimal system. In general, authors observed that the fewer rules participants had to internalize, the easier the execution of a problem-solving task was. Although Zhang and Norman [80] did not investigate the processes involved in dealing with program comprehension and fault fixing in source code, we can imagine that if people are accustomed to working with abbreviations, then the difference in their performance (e.g., effort) should not be so high when identifiers are in full-word and abbreviated forms.

In computer programming the use of abbreviations is not new (e.g., the abbreviation *tmp* stands for *temporary* and is often used to indicate a temporary variable, while *i* stands for *iterator* and very often developers used it to indicate an iteration counter). Today the use of abbreviations is widely used in social media slang as well. In general, slang consists of a lexicon of non-standard words and phrases in a given language [8]. The use of these words and phrases is typically associated with a subversion of a standard language (e.g., English). Social media slang is a kind of slang that has become more prevalent since the early 2000s. Its diffusion is the result of the rise in popularity of social media (e.g., Facebook and Twitter), where a character limit for each message requires a more condensed manner of communication (e.g., 4ward and YOLO). If people daily work with abbreviated identifiers and more and more use social media terms to communicate, we can imagine that they become more comfortable in dealing with source code that includes identifiers in abbreviated form. To confirm or contradict this assumption empirical studies are needed. Our investigation goes in this direction in the sense that we are mainly interested to quantitatively and qualitatively study if the presence of abbreviated identifiers in source code has an effect on fixing the faults it might contain. Qualitative studies can be considered a necessary complement to quantitative investigations [65] since they are essential for gaining an understanding of the reasons and motivations behind the problem under study (e.g., fault fixing when identifier names are either abbreviated or full-word). Our investigation (i.e., both the family of experiments and the qualitative study) is also explorative because it can be considered as a pre-study to a more thorough investigation on social media slang and abbreviated identifiers.

2.2 Replications in Software Engineering

The software engineering community has been embracing replications more readily (e.g., [14, 18, 38, 71]). In general, we can define a replication as the repetition of an experiment [5]. There are two factors that underlie the definition of a replication: *the procedure* (i.e., the steps to be followed)

and *the researchers* (i.e., who conducted the replication). As for the procedure, the kinds of replications range in between *conceptual* and *close*. A replication is conceptual if the research questions are the same, but the experimental procedure is different from that of a baseline experiment. On the other hand, a replication is close if its procedure is the same as that of the original experiment [71]. As for the researchers factor, researchers distinguish between *internal* and *external*. An internal replication was conducted by the same group of researchers as the baseline experiment [47], while an external replication is performed by different experimenters to avoid as much as possible experimenters' biases [23, 35, 71]. More recently, Gómez et al. [24] proposed a different classification for the types of replications based on the dimensions of experimental configuration that changes from a baseline experiment to a replication: *protocol*, *operationalizations*, *populations*, and *experimenters*. Based on changes to these four dimensions, the authors established three types of replications: *Literal*, where the aim is to run as exact a replication of the baseline experiment as possible; *Operational*, where the aim is to vary some (or all) of the dimensions of the baseline experiment configuration; and *Conceptual*, where experimenters have "nothing more than a clear statement of the empirical fact" [44].

Independently from the classification for types of replications, there are two primary motivations to perform replicate experiments in software engineering: (i) they are necessary to solve problems and to collect evidence because they bring credibility to a given research and (ii) they are valuable because they provide evidence on the benefits of a software engineering practice thus allowing industrial stakeholders to use this information to support adoption decisions [4, 16, 24, 51]. Whatever the kind of replication, a family of controlled experiments contribute to the conception of important and relevant hypotheses that may not be suggested by an individual experiment and/or single replications [5].

2.3 Dealing with Identifier Names

Source code comprehensibility and modifiability have been largely investigated [26, 56, 62, 63, 79]. Researchers studied the developer behavior during the execution of maintenance tasks [19, 41]. For example, LaToza et al. [41] conducted a qualitative study on developers focusing on their work habits: development, maintenance, and communication. Surveys and interviews were used to gather information from the participants in the study. As far as maintenance is concerned, the authors observed that developers remained focused on the code itself despite the availability of design documents, so concluding that documentation is inadequate for maintenance tasks. DeLine et al. [19] conducted an observational study on a few professionals who updated an unfamiliar implementation of a video game. Similar to LaToza et al. [41], developers considered inadequate the documentation for maintenance and evolution tasks. The studies before and that presented by Singer et al. [74] obtained a common result: developers prefer source code while dealing with program comprehension and maintenance tasks. This is why we focused our investigation on the use of source code. Summarizing, developers considered inadequate design and source documentation and tools when comprehending and evolving unfamiliar source code (e.g., [41, 54]). Therefore, the most appropriate strategy seems to deal only with source code (e.g., [56]).

Although source code seems the most important source of information for developers, researchers have marginally investigated the effect of kind and style of identifier names on source code comprehensibility and modifiability [11, 42]. For example, Binkley et al. [11] presented a family of studies to investigate the impact of two identifier styles (i.e., *camel case* and *underscore*) on comprehension participants achieved on source code. These studies involved 150 participants with varied demographics from two universities and considered both time and accuracy of comprehension. 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. Differently, Lawrie et al. [43] conducted a quantitative study with 100 programmers to investigate whether source code comprehension is affected 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 participants on these functions was assessed through questionnaires. Results showed that full-word identifiers lead to the best comprehension with respect to abbreviated identifiers, even if this difference is not significant. There are several differences between this study and our family of experiments. In particular, we focus on fault fixing: participants were asked to fix faults in source code, and finally fault fixing was performed on Java and C software downloaded from the Web. We also conducted a qualitative study as a complement to this family of experiments. This further study allowed us to better understand the values, beliefs, and assumptions that inform and shape fault fixing. We can state that our family of experiments fills a gap in the studies discussed just before (i.e., [42, 43]) because we have investigated the role of full-word and abbreviated identifiers on fault fixing tasks.

2.4 Qualitative Studies in Software Evolution and Ethnography

Qualitative studies appear to be still unusual in the software engineering field even if they are considered a necessary complement to quantitative studies [65]. For example, Roehm et al. [56] carried out a qualitative study in the field of program comprehension. One of the goals of this study was to understand how developers practice program comprehension and which methods and tools proposed by researchers are used in the software industry. Roehm et al. [56] found that developers put themselves in the role of end-users whenever possible. The authors observed developers inspecting the behavior visible in user interfaces and comparing it to the expected behavior. This strategy aims at understanding program behavior and getting first hints for further program exploration. Source code is considered the only source of information to perform comprehension tasks. In addition, results show a gap between program comprehension research and practice as the authors did not observe any use of state-of-the-art comprehension tools and developers seem to be unaware of them. On the other hand, Robillard et al. [54] conducted a qualitative (exploratory) study in the context of software maintenance and evolution. The authors studied five developers in a laboratory setting, while they have been updating unfamiliar source code. The developers were asked to implement a change request. The results supported the intuitive notion that a methodical and structured approach is the most effective.

Ethnography is a qualitative research method to study people and cultures. It is largely adopted in disciplines outside software engineering and in different areas of computer science [17, 28, 66]. Despite its potential, little ethnographic research exists in the field of software engineering [68]. In this context, ethnography could be very useful to provide an in-depth understanding of the socio-technological realities surrounding everyday software development practice [68]. Ethnography could also help to uncover not only what software engineers (e.g., novice and senior) do, but also why they do it. In fact, ethnographic studies are better suited to ask questions such as *how*, *why*, and *what are the characteristics of* [55]. The researcher attends to the taken-for-granted, accepted, and un-remarked aspects of practice, considering all activities as “strange” so as to prevent the researchers’ own backgrounds from affecting their observations [70]. To this end, the researcher immerses himself/herself in the environment where participants work and participates in the study, while collecting data by means of contemporaneous field notes, audio recordings, and copies of various artifacts. Ethnographic studies should take place on a long term. In some fields of research, such as software engineering, this practice is not always possible (e.g., because of time constraints). In these cases, it is common to adapt ethnographic methods to a shorter time frame [55] and on a small number of participants [69].

In the context of software maintenance and evolution field, Salviulo and Scanniello [60] conducted an ethnographically informed study with students and professionals. The goal of this study was to understand the role of comments and identifiers in source code comprehensibility and maintainability. Authors observed the following outcomes: (i) professional developers (as compared with students) prefer to deal with source code and identifiers rather than comments and (ii) the participants (professionals and students) believed essential the use of naming convention techniques when writing identifiers. There are differences and similarities between that study and the ethnographically informed study presented in this article. The most important difference is that we focus here on fault fixing (on code containing either abbreviated or full-word identifier names) that is a very specific kind of task in the context of software maintenance and evolution. Prior to fixing a fault, developers must inevitably investigate the source code of the target application. This aspect can be considered similar in both these studies although the differences in the experimental objects, e.g., the used code contained either abbreviated or full-word identifier names in the investigation presented in this article. In this respect, Salviulo and Scanniello [60] observed that participants believed that the names of identifiers were important and that developers should properly choose them. From our quantitative results, we observed that how identifier names are written is not a major issue to fix faults in source code. In some sense, these results and those shown in Salviulo and Scanniello [60] show a sort of discrepancy between what the developers claimed to do and what they actually did when dealing with source code. Ethnography is a viable empirical method to detect and explain such kind of discrepancies [70] and this is why we decided to conduct such kind of study to better analyze and explain the results from our family of experiments.

Singer et al. [74] studied how software engineers maintain a large telecommunications system. They focused on the developer's habits and tool usage during software development. This study was hosted in a single company. Also, Singer et al. [74] discovered a discrepancy between what developers claimed to do when performing maintenance operations and what they actually did. For example, despite the fact that developers stated that "reading documentation" was what they did, the study showed that searching and looking at source code was much more common than looking at documentation. Again, one of the values of ethnographic research is that it might help in highlighting and explaining such kind of discrepancies to make clearer un-remarked aspects of practice [70].

Beynon-Davies [9] conducted an ethnography study and observed that ethnographic research may be useful for capturing knowledge about intangible or unquantifiable aspects of the software lifecycle. In particular, these authors noted that for researchers in the software engineering field, ethnographic research may provide value in the area of software development, specifically in the process of capturing tacit knowledge during the software development. Later, Beynon-Davies et al. [10] used ethnography on rapid application development to uncover the negotiated order of work in a software project and the role of collective memory in software development.

Sharp and Robinson [69] reported a study on eXtreme Programming. This study was carried out in a small company that mostly based its business on the development of web-based intelligent advertisements. The main result the authors observed was that the XP developers were clearly "agile." In particular, Sharp and Robinson [69] observed that this agility seemed intimately related to the relaxed, competent atmosphere that pervaded the developers working in groups. Later, Romano et al. [58] conducted an ethnographically informed study in the context of TDD (Test Driven Development). TDD is an iterative software development technique where unit tests are defined before production code. Developers repeat short cycles consisting of (i) writing a unit test for an unimplemented functionality or behavior; (ii) supplying the minimal amount of production code to make unit tests pass; and (iii) applying refactoring where necessary, and checking that all tests are still passed after refactoring [7]. In this study, the authors involved 14 novice software

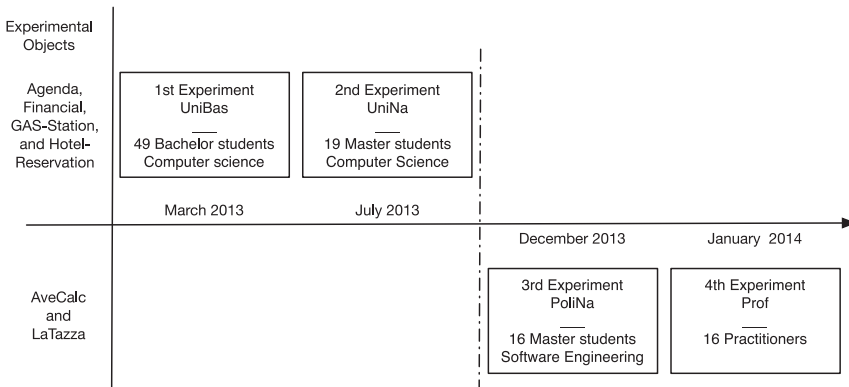


Fig. 1. Summary of the family of experiments.

developers (i.e., graduate students in Computer Science) and six professional software developers (with 1 to 10 years work experience). Romano et al. [58] concluded that developers (novice and professional) write quick-and-dirty production code to pass the tests and ignore refactoring.

3 THE FAMILY OF EXPERIMENTS

We conducted a family of controlled experiments composed of the baseline experiment, an internal closed replication, and two external replications that are also differentiated because different participants and experimental objects were used [5]. These replications are operational because we varied some dimensions of experimental configuration [24]. In PoliNa and Prof, we varied the experimenter, the population, and the operationalizations, while only the population in UniNa. This family is summarized in Figure 1. Rectangles represent experiments and are grouped by the experimental objects used. Agenda, Financial, GAS-Station, and Hotel-Reservation are all implemented in C, while AveCalc and LaTazza in Java. In Figure 1, we also summarize important aspects related to individual experiments: their execution order (e.g., the first experiment), the kind of participant (e.g., Bachelor students), the number of participants (e.g., 49), and the textual label associated to each experiment (e.g., UniBas).

We carried out our experiments by following recommendations provided by Juristo and Moreno [35], Kitchenham et al. [39], and Wohlin et al. [78]. We reported these experiments according to the guidelines suggested by Jedlitschka et al. [34]. For replication purposes, our experimental material (e.g., source code of used applications) is available online.⁴

3.1 Goal

We defined the main goal of our family of experiments that, by applying the Goal Question Metrics (GQM) template [6], can be formalized as follows: *Analyze the presence of either abbreviated or full-word identifiers in source code (e.g., C and Java) for the purpose of evaluating their effect with respect to effort, effectiveness, and efficiency to execute fault fixing tasks from the point of view of the researcher and the practitioner in the context of novice developers and students in Computer Science and Software Engineering unfamiliar with that code.*

According to this goal, we have defined and investigated the following two main research questions:

⁴www2.unibas.it/gscanniello/Identifiers/.

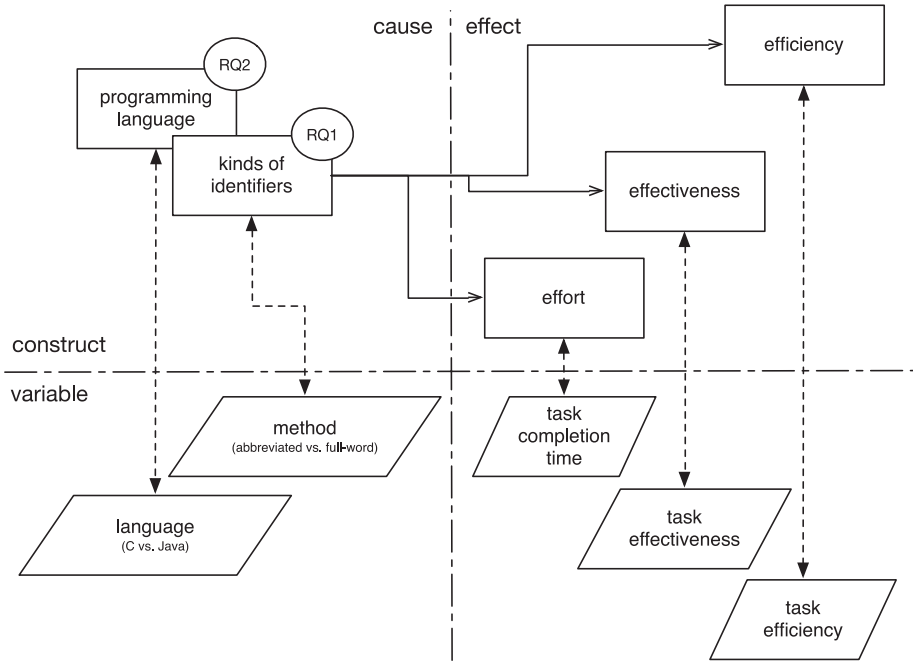


Fig. 2. Conceptual model.

RQ1: Does the presence of either abbreviated or full-word identifiers in source code penalize the effort to accomplish fault fixing tasks in that code and the effectiveness and efficiency to accomplish this kind of task?

RQ2: Does programming language (i.e., C and Java) affect the effort, effectiveness, and efficiency to execute fault fixing tasks?

When dealing with existing source code, a developer searches for relevant pieces of code (manually and using search tools) by following incoming and outgoing dependencies of relevant code and collects code and other information [40]. This could make indistinguishable program comprehension tasks and the identification and the fixing of faults in source code. This is why we simply refer to all these tasks together as either fault fixing or fault fixing task.

We show the conceptual model of our family of experiments in Figure 2. We built it on the basis of the formalization of our family of experiments, our own experience, and an analysis of the literature about source code comprehension and approaches to write identifier names [11, 42, 43, 60]. The relationship between our conceptual model and both RQ1 and RQ2 is also shown. For example, the bottom part (right-hand side) of this figure shows the metrics we exploited to measure the constructs: effort, effectiveness, and efficiency in fixing faults in source code. We studied the same relation between cause and effect for both RQ1 and RQ2.

We also defined and investigated two additional research questions. We consider these questions secondary because they are not directly related to the main goal of our study. This is why we do not include them in the conceptual model in Figure 2. These further research questions are as follows:

RQ3: Can developers correctly and completely expand abbreviated identifiers?

RQ4: Do developers perceive fault fixing on source code containing abbreviated identifiers as more difficult/simple than fault fixing on source code containing full-word identifiers?

We investigated RQ3 to understand the possible difficulties developers have in associating the right meaning to the abbreviated compounding words of identifiers. We investigated RQ4 to understand if developers perceive abbreviated identifiers in source code harmful (or not) when they have to fix faults in that code. The presence of abbreviated identifiers is common in source code and this trend has been becoming even more evident in recent years [3].

3.2 Context Selection

The applications/programs considered in our family of experiments are summarized in Table 1. In this table, we also report some descriptive statistics on these programs: Lines of Code (LOC), number of files or classes (#files/#classes), and percentage of abbreviated identifiers. As for LOC, we considered source code without comments. Reported source code metrics have been gathered by the Understand tool.⁵ The percentage of abbreviated identifiers is computed with respect to the number of full-word identifiers in the original versions of the applications. The number of abbreviated identifiers was manually counted. The percentage of abbreviated identifiers in the applications seems to confirm our assumption that the use of identifier names in abbreviated form is common in software development.

The source code of the Java and C applications contained both abbreviated (e.g., `mobNum` in Agenda) and full-word identifiers (e.g., `filePointer` in GAS-Station). Acronyms were also present (e.g., CFU indicated `CreditFormativeUnit`). One of the authors (the second) had to create two versions for each application. One version contained only abbreviated identifiers, while the other only full-word identifiers. The author expanded compounding words of each identifier using the Babel⁶ dictionary. Full-word and abbreviated identifiers were both in English. Source code comments were omitted from each created version. The presence of comments could be an extraneous factor and could affect participants' results in an undesirable way (e.g., the effort to fix faults). The source code layout was similar in C and Java programs (e.g., statements were semantically grouped).

We downloaded from the Web the C applications used in our baseline experiment and in UniNa. Similarly, the Java applications used in PoliNa and Prof were distributed online and have been already employed in a number of empirical studies (e.g., [53]). We chose both the C and Java applications because their application domains can be considered a good compromise of generality and industrial application. We also selected the two Java applications so that they were not much different from one another. Similar considerations can be done on the C applications used in UniBas and UniNa. This is advisable in software engineering experiments [78].

For each application, we injected the same faults in each variant (i.e., with only abbreviated identifiers and with only full-word identifiers) by performing mutation analysis [20]. All the versions of the applications used in our family of experiments were syntactically correct.

Mutation analysis was originally introduced as a method to evaluate a test suite in how good that suite is at detecting faults. The main idea behind this kind of analysis is to seed artificial faults based on what is thought to be real errors commonly made by programmers [22]. This was the main reason why we used mutation analysis to inject faults in the applications used in our family of experiments.

The injection process was based on the Kim's mutation operators [36, 37]. These operators are specific to object-oriented languages and a few of them can be also applied to imperative programming languages. In Table 2, we show the Kim's operators (and their acronyms as defined by the author [36, 37]) and how many times we applied them to each application implemented in C. A short description of these operators is also shown. In Table 3, we show the Kim's operators, their

⁵<https://scitools.com>.

⁶www.cs.tut.fi/tlt/stuff/misc/babel.html.

Table 1. The Experimental Objects Used in the Family of Experiments

Experiments	Name	Description	LOC	#files/ #classes	% Abbr. Identifiers
UniBas & UniNa	Agenda	It keeps track of personal contacts. In particular, it allows the user to add new contacts by specifying their 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 contacts can be also sequentially browsed. The user can load a file containing contacts. This enables the user to manage different contact lists separately. http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=2299&lngWId=3 .	609	4/-	29.5%
	GAS-Station	It manages the main feature of a GAS station. The application takes as input the daily price of Petrol-Oil, Diesel, and Compressed Natural GAS. GAS-Station is able to manage receipts and bills regarding the refueling of Petrol-Oil, Diesel, and Compressed Natural GAS. The bills are stored on the hard-disk to be successively searched, modified, and browsed by the user. http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=11639&lngWId=3 .	649	1/-	37.2%
	Financial	It is a command line option price calculator. It uses Black-Scholes, which is a mathematical description of financial markets and derivative investment instruments. The model develops partial differential equations whose solution (i.e., the Black-Scholes formula) is widely employed to price European puts and calls on non-dividend paying stocks. http://www.paulgriffiths.net/program/c/finance.php .	202	3/-	37.8%
	Hotel-Reservation	It manages room reservations for Hotels. To reserve a room the application asks for the fiscal code number of the client and dates for both the check-in and check-out. The user can modify and delete reservations as well. The status of all the rooms in the hotel can be also shown in Hotel-Reservation. http://www.vyomworld.com/source/code.asp?id=01&l=C_projects&t=HotelReservationSystem .	458	1/-	50.0%
PoliNa & Prof	AveCalc	It is a desktop application written in Java that manages an electronic register (record book) for Master's students. A student can add a new exam to the register, remove an existing exam, and remove all exams. AveCalc computes some statistics: average of the exams passed, total number of CFUs (i.e., Credit Formative Units), number of exams passed, (hypothetical) degree vote, and whether the student has passed a number of exams sufficient to defend his/her thesis.	1,388	8/33	39.7%
	LaTazza	It is a desktop application for a hot drinks vending machine. LaTazza supports sale and supply of small-bags of beverages (Coffee, Tea, Lemon-tea, etc.) from the Coffee-maker. This application supports two kinds of clients: visitors or employees. Employees can purchase beverage paying by cash or on credit, visitors only paying by cash. The secretary can sell small-bags to clients, buy boxes of beverages, manage credit and debt of employees, check the inventory, and check the cash account.	1,215	18/18	72.2%

Table 2. Injected Faults in C Programs

Operator	Description	Agenda	GAS-Station	Financial	Hotel-Reservation
Literall Change Operator (LCO)	It increases/decreases numeric values or swap Boolean literals.	2	1	-	-
Language Operator Replacement (LOR)	It replaces a language operator with other legal alternatives.	1	1	1	-
Control Flow Disrupt operator (CFD)	It disrupts normal control flow: add/remove break, continue, and return statements.	1	1	-	1
Variable Replacement Operator (VRO)	It replaces a variable name with other names of the same type and of compatible types.	-	1	1	1
Total number of injected faults		4	4	2	2

Table 3. Injected Faults in Java Programs

Operator	Description	AveCalc	LaTazza
Language Operator Replacement (LOR)	It replaces a language operator with other legal alternatives.	-	1
Control Flow Disrupt operator (CFD)	It disrupts normal control flow: add/remove break, continue, and return statements.	1	1
Variable Replacement Operator (VRO)	It replaces a variable name with other names of the same type and of compatible types.	-	1
Relational Operator Replacement (ROR)	It replaces relational operators with other legal alternatives.	1	-
Assignment Operator Replacement (ASR)	It replaces shortcut assignment operators.	2	1
Arithmetic Operator Insertion (AOI)	It inserts shortcut arithmetic operators.	1	1
Conditional Operator Insertion (COI)	It inserts unary conditional operators.	-	1
Conditional Operator Replacement (COR)	It replaces binary conditional operators with other binary conditional operators.	1	-
Total number of injected faults		6	6

description, and how many times they have been applied on the Java applications. In Appendix A, we provide two examples of application of the Kim's operators used in the AveCalc and Hotel-Reservation applications.

While choosing mutator operators, we made the assumption (i.e., competent programmer hypothesis [20]) that very often source code is very close to its correct version, or that the difference between current and correct code for each fault is very small [25]. When applying mutation operators, our main objective was to create faults representative as real ones and avoid as much as possible interactions among seeded faults. In addition, we tried to apply operators that seeded faults that presumably had similar complexity to be fixed and that did not interact with the main factor under study. We performed an analysis post-hoc to verify these assumptions. The obtained results are summarized in Appendix B.

We conducted the experiments in research laboratories under controlled conditions. For each experiment, participants had the following characteristics:

UniBas: Participants were 2nd-year undergraduate students of a course on Algorithms and Data Structures. The experiment was conducted as an optional exercise in this course.

Participants had passed all the exams related to the following courses: Procedural Programming and Object Oriented Programming I. In these courses, participants studied C/C++ and Java on university problems.

UniNa: Participants were 2nd-year graduate students of a Software Engineering II course. The experiment was conducted as an optional laboratory exercise in this course. During the Bachelor program in Computer Science at the University of Naples, participants had passed all the exams related to the following courses: Procedural Programming, Software Engineering I, Object-Oriented Programming I and II, and Databases. Participants had a reasonable level of technical maturity and knowledge of software design, development, and testing.

PoliNa: Participants were 2nd-year graduate students at the Polytechnic of Naples. The experiment was conducted as an optional laboratory exercise of an Advanced Software Engineering course. Students had passed all the exams related to the following courses: Procedural Programming, Software Engineering I, Object-Oriented Programming I and II, and Databases. Participants had a reasonable level of technical maturity and knowledge on object-oriented software design, development, and software testing. Students' development experience level can be considered similar to that of participants in UniNa.

Prof: Participants all had a Master's degree. In particular, we had 10 Biomedical Engineers, 4 Telecommunication Engineers, and 1 Electronic Engineer. There was also one participant with a Master's degree in Computer Science. All the participants had passed exams on the following courses held in a professionalization program in the context of design of platforms for innovative services in the future Internet: Object Oriented Programming, Mobile Applications Development, Testing and Debugging Techniques. Participants in Prof also made an internship in industry. The experiment was conducted at the end of this internship. We can consider development experience of participants in Prof not inferior to that of graduate students, but superior to participants in UniBas.

Participants in UniBas, UniNa, and PoliNa were informed that their grade on the course in which the experiment was conducted would not be affected by their performance in that experiment. For their participation in the study, we rewarded them with a bonus in their final mark. Participants in Prof participated in this experiment as part of their professionalization program. This choice was made to encourage participation. In addition, we did not compel them to participate. We informed participants that data collected during the experiment were only used for research purposes, treated confidentially, and shared in aggregated form and anonymously.

3.3 Variable Selection

The bottom part of Figure 2 reports the metrics used to assess the effect of having full-word or abbreviated identifiers on fault fixing. Participants who were given source code without comments and with full-word identifiers comprised the *control group*, while the *treatment group* comprised participants who were given source code without comments and with abbreviated identifiers. Thus, *method* is the main independent variable of our study, or also main factor or manipulated factor or main explanatory variable, from here on. This variable is nominal and assumes values ABBR (source code without comments and with abbreviations) and FULL (source code without comments and with full-word identifiers). For each application, the source code layout was the same in both the variants (i.e., ABBR and FULL). We also investigate if the programming language with which the experimental objects were implemented (i.e., *language*) might affect fault fixing. Thus, *language* is a nominal variable and assumes C and Java as the possible values.

To study RQ1 and RQ2, we considered the effort, the effectiveness, and the efficiency to fix faults in source code. For each of these constructs, we used a measure. In the following subsections, we describe these measures and their link with the constructs (see Figure 2).

3.3.1 Effort and Task Completion Time.. To assess the effort construct, we used task completion time. Considering the time as an approximation for effort is customary in literature and it is compliant with the ISO/IEC 9126 standard [31] definition: effort is the productive time associated with a specific project task. Other aspects that may be related to effort—for example, the cognitive effort of participants—were not measured. Task completion time indicates the number of minutes to accomplish a fault fixing task and then it assumes only positive integer values. Low values for task completion time mean that participants spent less time (or less effort) to complete a fault fixing task.

3.3.2 Effectiveness and Task Effectiveness.. We considered the number of faults successfully identified and removed from the source code. This construct takes into account correctness and completeness of fault fixing tasks. We measured correctness and completeness by using an information retrieval-based approach [59]. In particular, we estimated correctness by the precision measure (P), while completeness by recall (R). We used the following formulations for these two measures:

$$P = \frac{\#(\text{faults correctly fixed})}{\#(\text{faults fixed})}, \quad (1)$$

$$R = \frac{\#(\text{faults correctly fixed})}{\#(\text{faults present})}. \quad (2)$$

$\#(\text{faults fixed})$ includes bugs that the participant both correctly and incorrectly fixed. We could easily compute this value because we knew the source code without seeded faults, the source code with injected faults, and the source code modified by participants at the end of each experimental trial. We determined if a fault was correctly or incorrectly fixed in two stages. First, for each participant and fault, we inspected the modified source code. If this modification produced the same source code as before applying mutation operators, we concluded that the fault was correctly fixed. Otherwise, we applied test cases on the participant source code. If all test cases passed, we assumed that the participant had correctly fixed the fault. On the original source code of each application (before applying mutation operators), we defined from three to five test cases (input and expected output). These test cases were used to perform a sort of regression testing on the applications modified by the participants. The rationale behind this second step was that there is not a single way to fix a bug. It is worth mentioning that we provide no test case to the participants to perform fault fixing in the experimental trials.

As for precision, it assumes value 1, if and only if all the faults a participant fixed were correctly fixed. For example, if a participant fixes two faults (out of four) in Agenda and these two faults are correctly fixed, then precision assumes 1 ($\#(\text{faults fixed}) = 2$ and $\#(\text{faults correctly fixed}) = 2$) as the value. On the other hand, if a participant fixes four faults in Agenda and among them only two are correctly fixed, then precision assumes 0.5 ($\#(\text{faults fixed}) = 4$ and $\#(\text{faults correctly fixed}) = 2$) as the value. This measure assumes 0 as the value, when the participant either did not fix any faults or no fault was correctly fixed. Recall is 1 when all faults we injected were correctly fixed. For example, if a participant fixes four faults (out of four) in Agenda and all of them are correctly fixed, then recall assumes 1 ($\#(\text{faults correctly fixed}) = 4$) as the value. In all the other cases, recall assumes a value less than 1. In the case of the C programs, we used the following aggregated formulations for precision and recall: $P = \frac{\sum_i \#(\text{faults correctly fixed in application}_i)}{\sum_i \#(\text{faults fixed in application}_i)}$ and $R = \frac{\sum_i \#(\text{faults correctly fixed in application}_i)}{\sum_i \#(\text{faults present in application}_i)}$. Aggregated formulations for precision and recall assume

values in between 0 and 1 and can be interpreted as the P and R shown in Equations (1) and (2), respectively. These different formulations for P and R were needed because participants in UniBas and UniNa were asked to fix faults in two C applications in each experimental trial (see Section 3.5).

A single measure that trades off precision versus recall is the F -measure [46]. It is a weighted harmonic mean of precision and recall. In our case, we opted for the default balanced F -measure (commonly written as F_1). It equally weights precision and recall:

$$F_1 = \frac{2 * P * R}{P + R}. \quad (3)$$

F_1 assumes values in between 0 and 1 and measures *task effectiveness*. For this measure, a value equal to 1 means that a participant correctly and completely removed all the faults. That is, the participant obtained 1 as the value for both precision and recall. In all the other cases, F_1 assumes a value less than 1.

3.3.3 Efficiency and Task Efficiency. We also considered the efficiency with which a fault fixing task was accomplished. The efficiency construct is measured by means of the variable *task efficiency*. It is a derived measure that is computed as the ratio between task effectiveness and task completion time and estimates the efficiency of a participant in the execution of a fault fixing task. That is, this measure estimates the ability to effectively fix faults in source code without wasting time:

$$task\ efficiency = \frac{task\ effectiveness}{task\ completion\ time}. \quad (4)$$

The larger the task efficiency value, the better it is. The perspective we adopted is that of the quality in use (e.g., [32, 33]): efficiency measures task effectiveness achieved to expenditure of resources. For example, if a participant $p1$ obtains 1 for task effectiveness and 25 minutes for task completion time, then task efficiency is equal to 0.04. On the other hand, if a participant $p2$ obtains 0.8 for task effectiveness and 30 minutes for task completion time, then task efficiency for this participant is equal to 0.0266. Comparing the task efficiency values for $p1$ and $p2$, we understand that $p1$ is better because he/she has got a better tradeoff between effectiveness and effort to identify and fix faults.

3.4 Hypotheses Formulation

The following null hypotheses have been formulated and tested.

- Hn0:** The method (ABBR or FULL) does not affect the effort (task completion time) to fix faults.
- Hn1:** The method does not affect the effectiveness (task effectiveness) when performing fault fixing.
- Hn2:** The method does not affect the efficiency (task efficiency) when performing fault fixing.
- Hn3_X:** There is no difference in fixing faults in the source code written either in C or Java with respect to the dependent variable X (i.e., task completion time, task effectiveness, and task efficiency).

Alternative hypotheses for Hn0 ($\neg Hn0$), Hn1 ($\neg Hn1$), and Hn2 ($\neg Hn2$) admit a positive effect of ABBR or FULL. All these null hypotheses are two sided because we could not postulate any effect of method on the dependent variables (or also response variables, from here on). We defined them to investigate the research question RQ1, while Hn3 (described as a single parameterized hypothesis) concerns the investigation of RQ2.

Table 4. Experiment Design

Trial	Variable	Group A	Group B	Group C	Group D
First	Object	A+F or AveCalc	A+F or AveCalc	G+H or LaTazza	G+H or LaTazza
	Method	ABBR	FULL	ABBR	FULL
Second	Object	G+H or LaTazza	G+H or LaTazza	A+F or AveCalc	A+F or AveCalc
	Method	FULL	ABBR	FULL	ABBR

Table 5. Group Composition

Experiment	Group A	Group B	Group C	Group D	Total
UniBas	13	13	12	11	49
UniNa	5	5	4	5	19
PoliNa	4	4	4	4	16
Prof	4	4	4	4	16

3.5 Design of the Experiments

We used the experimental design summarized in Table 4. It ensured that each participant works on different experiment objects in two laboratory trials/runs, using ABBR or FULL each time. As for UniBas and UniNa, the experimental objects were *A+F* (i.e., Agenda plus Financial) and *G+H* (i.e., GAS-Station plus Hotel-Reservation). We grouped the selected applications in experiment objects according to their size, number of injected faults, and application domain. For each experimental object, we provided participants with the source code of two applications forming the experimental object together. Therefore, given a laboratory trial participants could freely work on any fault and application. As for PoliNa and Prof, we replaced the objects *A+F* and *G+H* with *AveCalc* and *LaTazza*, respectively.

We opted for this kind of design because it mitigates possible carryover effects⁷ as well as the effect of experimental objects in each experiment [78]. In UniBas, UniNa, and PoliNa, we used participants' average grades to distribute participants among groups in Table 4. In other words, these groups were similar to each other in terms of number of participants with high and low average grades.⁸ As for participants in Prof, Master's degree grade⁹ was used to distribute them among groups in Table 4. For each experiment of our family, we report the number of participants assigned to each of these groups in Table 5. It is worth mentioning that we originally assigned 12 students to the group D, but because of a health problem one of these students had to abandon the first trial in UniBas.

3.6 Experimental Tasks

We asked participants to perform the following tasks:

- *Fault Fixing*. We asked to identify and fix faults in the source code of the experimental objects according to our experimental design (see Table 4). For each experimental object,

⁷Carryover is the persistence of the effect of one treatment when another treatment is applied later [76].

⁸In 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, and high otherwise.

⁹In Italy, grades are expressed as integers and assume values in between 66 and 110 cum laude. We consider the grade as low if it is below 105/110, and high otherwise.

ID	2
Title	Truncated domain in e-mail address
Description	In some cases the domain of the e-mail address is truncated when contacts are shown (both browsing and searching for). For example, when the e-mail address giuseppe.scanniello@unibas.it was inserted the application did not produce any warning message, while when I searched it Agenda showed giuseppe.scanniello@uniba followed by birthday.

Fig. 3. A sample of bug report for the Agenda application.

we provided participants with its mission (or problem statement). A mission summarized the intent of an application. We also gave participants a bug report for each injected fault. A bug report contained both the title of the bug (i.e., a short description) and its (long) description and ID. Given a bug, its report is the same independently from the method (ABBR and FULL). For each fault, the participants had to indicate the portion of modified source code that fixed that fault (i.e., the patch). In particular, the participants had to delimit the code of the patch using comments as follows: `/* fault <fault_ID> start */` at the beginning of the patch and `/* fault end */` at the end of the patch. Since we knew the injected faults, the number of faults a participant correctly fixed was easy to compute. This information was necessary to estimate the effectiveness and efficiency constructs.

In each bug report, there was no reference to the source code, namely, identifiers in abbreviated and full-word form were not mentioned. In addition, we wrote reports trying to give the same level of detail and accuracy in each bug description. This is not a major issue here because possible biases related to unbalances in bug descriptions would equally affect tasks accomplished with both ABBR and FULL. A bug report for the Agenda application is shown in Figure 3.

- *Expanding Identifiers.* We asked participants to expand abbreviated identifiers in the source code of the task accomplished with ABBR. In particular, we gave the complete list of abbreviated identifiers contained in the code in a separate file (i.e., a CSV file) and asked them to write for each abbreviated identifier its full-word version. We did not impose any rule and time limit to expand identifiers. We gave this list at the end of the second laboratory trial. The goal of this task was to assess whether or not participants correctly expanded abbreviated identifiers. For each identifier, we had both its abbreviated and full-word versions. A participant correctly expanded an abbreviated identifier name if he/she provided its full-word version. We used the data gathered from the expanding identifiers task to study RQ3. Please note that task completion time (i.e., the variable to assess the effort construct) does not take into account the time needed to expand abbreviated identifiers.
- *Post-experiment task.* We asked participants to fill in the post-experiment survey questionnaire shown in Table 6. The statements of the post-experiment survey questionnaire admit closed answers according to a five-point rating scale: (1) strongly agree; (2) agree; (3) neutral; (4) disagree; and (5) strongly disagree. The goal of that questionnaire was to obtain feedback about participants' perceptions of the experiment execution and about the perceived effect of abbreviated identifiers on fault fixing tasks. For example, even if we did not impose any time limit to perform laboratory trials, we asked Q1 to understand if participants perceived any pressure to accomplish the tasks. Statements Q5 and Q6 in Table 6 were exploited to study the research question RQ4. In UniNa, PoliNa, and Prof, we asked participants questions on the used social media and slang. In addition, we also asked questions to understand if participants perceived that social media slang was changing how developers chose identifiers and wrote source code comments. The added questions are those reported in Table 7. Since the participants might have a different notion of social-media slang, we provided them with a definition of social-media slang and some well known terms of such slang.

Table 6. Post-Experiment Survey Questionnaire Administered to Participants in UniBas

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	I found it difficult to understand the abbreviated identifiers.
Q5	The presence of abbreviated identifiers made the fixing of faults in source code difficult.
Q6	There is no difference in using abbreviated or full-word identifiers while identifying and fixing faults in source code.
Q7	I find the execution of fault fixing tasks useful from a practical point of view.

Table 7. Additional Questions to the Post-Experiment Survey Questionnaire Administered to the Participants in UniNa, PoliNa, and Prof

Id	Question
Q8	Indicate the social media you most frequently use.
Q9	I use social media: <input type="checkbox"/> every day <input type="checkbox"/> often <input type="checkbox"/> rarely <input type="checkbox"/> never
Q10	I use social media slang to communicate through social media <input type="checkbox"/> every day <input type="checkbox"/> often <input type="checkbox"/> rarely <input type="checkbox"/> never
Q11	I think that the social media slang is changing how developers choose identifiers. <input type="checkbox"/> yes <input type="checkbox"/> no
Q12	I think social media slang is changing how developers write source code comments. <input type="checkbox"/> yes <input type="checkbox"/> no

3.7 Experiment Operation

All the participants in our family of experiments first attended an introductory lesson (a few days before the experiment) in which we presented detailed instructions on the experiment procedure. Details on experimental hypotheses were not provided to avoid affecting results.

We asked participants to accomplish a training exercise similar to that which would appear in actual experimental tasks. As for UniBas and UniNa, we used *Tower of Hanoi*.¹⁰ This application was originally implemented in C. We ported this version from C to Java and used the latter in PoliNa and Prof as a training exercise. We injected the same two bugs in the C and Java versions of the Tower of Hanoi. The rationale behind this design decision was to allow participants to familiarize themselves with the experimental procedure that they then used in the laboratory trials.

After the introductory lesson, but on the same day as the training exercise, the participants filled in a pre-questionnaire. We asked questions to understand whether there were any inherent differences among participants in each experiment and to get information on their university career

¹⁰<http://www.paulgriffiths.net/program/c/hanoi.php>.

(e.g., average grades of passed exams for students and Master's degree grades for participants in Prof). We used the gathered data to assign participants to the groups A, B, C, and D in Table 4.

To surf, execute, and debug source code, participants in UniBas and UniNa used Dev-C++¹¹ (release 4.9.9.2) in both the training exercise and the experiment. This is an open source Integrated Development Environment (IDE) for C/C++. We opted for this IDE because it was widely used in academic programming courses. As for PoliNa and Prof, participants used Eclipse in the training exercise and experiment. We opted for this IDE because it was well known and widely used in academia and industry.

We asked participants in our family of experiments to use the following experimental procedure: (i) specifying their name and start-time; (ii) identifying and fixing the faults; and (iii) marking the end-time. Participants could compile and execute the applications before and after each modification performed on source code. We did not suggest any approach to surf, execute, and debug source code. That is, participants could freely use the features implemented in the used IDE. This design choice allowed us to reproduce what real developers do when dealing with faults fixing. That is, this allowed us to reduce possible external validity threats.

To carry out the experiment, participants first received the material of the first laboratory trial. For example, we provided participants in group A of UniBas with the source code of both Agenda and Financial and corresponding bug reports. Participants freely chose how to deal with faults in these applications. When participants had finished the first trial, the material for the second laboratory trial was provided to them. After the completion of both the trials, we asked participants to expand abbreviated identifiers in the source code of the task accomplished with ABBR. Once participants concluded this task, we gave them the post-experiment questionnaire and asked them to fill it in. Interaction was permitted among participants neither while accomplishing tasks nor while passing from the first trial to the second one.

3.8 Analysis Procedure

To perform data analysis, we used the *R environment*¹² for statistical computing and we carried out the following steps:

- (1) We undertook the descriptive statistics of the dependent variables.
- (2) To test Hn0, Hn1, and Hn2, we applied multivariate linear mixed model analyses. This kind of model is an extension of the general linear model and it is a better method for analyzing models with random coefficients (as is the case of participants in software engineering experiments) and data dependency due to repeated measures (as is the case of our experiments) [76]. We used a multivariate linear mixed model analysis to verify the effect of the main factor/variable (i.e., Method) and other explanatory variables (i.e., Experiment, Trial, and Object) and the presence of a significant interaction¹³ between the main factor and these explanatory variables. We also applied a multivariate linear mixed model to test Hn_X. To apply multivariate linear mixed model analyses two assumptions have to be verified: that residuals follow a normal distribution and their mean has to be equal to zero [76]. In the absence of normality of residuals, transformation of the response variable data is an option (e.g., exponential and logarithmic transformations). We used the Shapiro-Wilk W test [67] (Shapiro test, in the following) to perform the normality analysis

¹¹<http://www.bloodshed.net/dev/>.

¹²www.r-project.org.

¹³The presence of an interaction makes it more difficult to predict consequences of one variable with respect to another one given a factor.

Table 8. Summary of Differences Between UniBas and Its Replications

	UniNa	PoliNa	Prof
Kind of participants	≠	≠	≠
Experimental objects	=	≠	≠
Post-experiment task	≠	≠	≠
Dependent variables	≠	≠	≠
Experimenters	=	≠	≠

of residuals. A p-value smaller than an α value allows concluding that residuals are not normally distributed.

- (3) To summarize and analyze raw data and to support their discussion, we exploited different graphical representations, such as boxplots and clustered bar charts [50, 78].

In all the executed statistical tests, we decided (as is customary) to accept a probability of 5% (i.e., α value) of committing Type-I-error [78].

3.9 Summary of the Differences

We introduced a number of differences between the baseline experiment by Scanniello and Risi [64] and its replications. These differences are sketched in Table 8 and summarized in the following:

- Participants in UniNa, PoliNa, and Prof can be considered more experienced than those in UniBas. This alteration was made to reduce external validity threats and to better analyze the effect of the kind of participants on the studied constructs.
- We changed the experimental objects in PoliNa and Prof. In these two experiments, participants were asked to fix faults in Java source code rather than in C source code. This alteration has also the value of reducing external validity threats.
- We extended the post-experiment survey questionnaire and administered it to participants in all the replications. This alteration was made to study RQ4.
- We considered new dependent variables. We introduced this alteration because these new variables are considered basic measures for information retrieval to assess effectiveness and efficiency [46]. These measures have been also largely used in empirical software engineering (e.g., [1, 52]). In addition, we believe that task effectiveness and task efficiency better fit the constructs object of our family of experiments.
- We extended and modified the data analysis. This modification is a direct consequence to the modification just before.
- A different experimenter conducted both PoliNa and Prof.

3.10 Threats to Validity

To better comprehend strengths and limitations of our family of experiments, threats that could affect results and their generalization are presented and discussed in this section. Despite our effort to mitigate as many threats as possible, some of them are unavoidable. We discuss the threats to validity following the guidelines proposed by Wohlin et al. [78].

3.10.1 Internal Validity. Threats to this kind of validity are influences that can affect the independent variable with respect to causality.

- *Maturation*. The adopted experimental design (a kind of crossover design—Table 4) should mitigate the presence of carryover effects [78]. We also analyzed the effect of the laboratory trials on the response variables. If this effect is present, we will properly mention it in our data analysis.
- *Diffusion or imitation of treatments*. This threat concerns information exchanged among participants within each experiment and among experiments. Experiment supervisors monitored participants to prevent that they communicated with one another. As an additional measure to prevent diffusion of material, we asked participants to return material at the end of each trial.
- *Selection*. The effect of letting volunteers take part in an experiment may influence results. In fact, volunteers could be more motivated.

3.10.2 External Validity.. External validity threats concerns generalizability of results.

- *Interaction of selection and treatment*. The use of students as participants may affect generalizability of results [13, 15, 29]. To mitigate this kind of threat, we conducted replications with novice professional software developers. A qualitative study with professionals was also conducted to deal with this kind of external validity threat.
- *Interaction of setting and treatment*. In our study, the kind of experimental tasks may affect result validity. For example, differences observed on the kind of experimental objects could not be related to the C and Java programming language, but to extraneous factors (e.g., the domain of the applications). Also the size and the complexity of used applications (both those implemented in C and Java) might affect external validity. The use of Dev-C++ in UniBas and UniNa and the use of Eclipse in PoliNa and Prof might also threaten the validity of results. For example, it could be possible that a few participants were more familiar with Eclipse than others. Identifier renaming (when producing the two versions of each application) might also bias results. Another threat to external validity is that identifiers and bug reports were in English, while participants were Italian students and professionals. That is, the participants' skill in English could be an issue if they are not at an advanced level. Injected faults might also threaten external validity. In this respect, our design choice reduced possible *experimenters' expectancies* (see next section) even if in this way we introduced issues related to representativeness of seeded faults with respect to real ones.

3.10.3 *Construct Validity..* Construct validity threats regard the link between concrete experimental elements and abstract concepts of experimental goals. Threats to construct validity are also related to experiment design and social factors.

- *Interaction of different treatments*. To mitigate this kind of threat, we adopted a kind of crossover design and analyzed the possible effect of co-factors.
- *Confounding constructs and level of construct*. The procedure used to divide participants into the groups in Table 4 could affect construct validity.
- *Experimenters' expectancies*. To deal with this kind of threat we used an injection process based on the Kim's mutation operators [36, 37]. In addition, faults were injected to favor none between ABBR and FULL. The post-experiment survey questionnaire was designed to capture participants' perception and designed using standard methods and scales [49].

3.10.4 *Conclusion Validity..* It concerns issues that may affect the ability of drawing correct conclusions.

- *Random heterogeneity of participants*. There is always heterogeneity in a study group. If the group is very heterogeneous, there is a risk that the variation due to individual differences

Table 9. Descriptive Statistics Task Completion Time, Task Effectiveness, and Task Efficiency

Experiment	Variable	FULL		ABBR	
		Mean	St. Dev.	Mean	St.Dev.
UniBas	task completion time	73.4898	33.3979	78.3673	33.1239
	task effectiveness	0.7203	0.1893	0.7277	0.1804
	task efficiency	0.0137	0.0126	0.012	0.0085
UniNa	task completion time	38.8947	32.7633	38.1053	29.0438
	task effectiveness	0.8186	0.2484	0.7709	0.2688
	task efficiency	0.0365	0.027	0.033	0.0228
PoliNa	task completion time	47.6875	25.1123	45.6875	16.1275
	task effectiveness	0.5541	0.2392	0.5184	0.2356
	task efficiency	0.016	0.0101	0.0116	0.0058
Prof	task completion time	47.9375	28.5971	50.125	15.2003
	task effectiveness	0.5672	0.1834	0.548	0.213
	task efficiency	0.0156	0.0097	0.012	0.0063

is larger than that due to the treatment [78]. As far as Prof, experience of participants could be heterogeneous.

- *Reliability of treatment implementation.* A possible threat concerns the fact that we did not impose any time limit to perform tasks. It could be possible that the experiments were not able to reveal differences because of this design choice. As for correctness and completeness of the expansions of identifiers, our design choice could have affected results. In particular, participants who fixed faults with ABBR in the second trial could be advantaged with respect to those participants experimented ABBR in the first trial. However, if we would ask participants to expand identifiers at the end of the first trial we had the issue of overburdening participants and then affecting results in the second trial. Participants performed fault fixing tasks on either Java or C source code. This could also affect the validity of the obtained results. The IDEs and their tools (e.g., debugger and search engine) could have also affected conclusion validity, when comparing UniBas and UniNa with PoliNa and Prof.
- *Reliability of measures.* This threat is related to how response variables were measured. For example, faults could not be equally difficult to find and to fix in each application and across the applications (see Appendix B, where we present the results of a further analysis on the faults seeded in the applications used as experimental objects). Then, the method used to assess effectiveness and efficiency constructs could have affected results. Regarding task completion time, we asked participants to write start and stop times (e.g., [30]). The measures used to estimate correctness and completeness of the identifier expansion (see Section 4.3) could affect results concerning the research question RQ3.

4 RESULTS FROM THE FAMILY OF EXPERIMENTS

In this section, we present the results of our data analysis following the procedure previously presented in Section 3.8.

4.1 Descriptive Statistics and Exploratory Data Analysis

In Table 9, we report the values of mean and standard deviation for task completion time, task effectiveness, and task efficiency grouped by method and experiment. The distribution of the

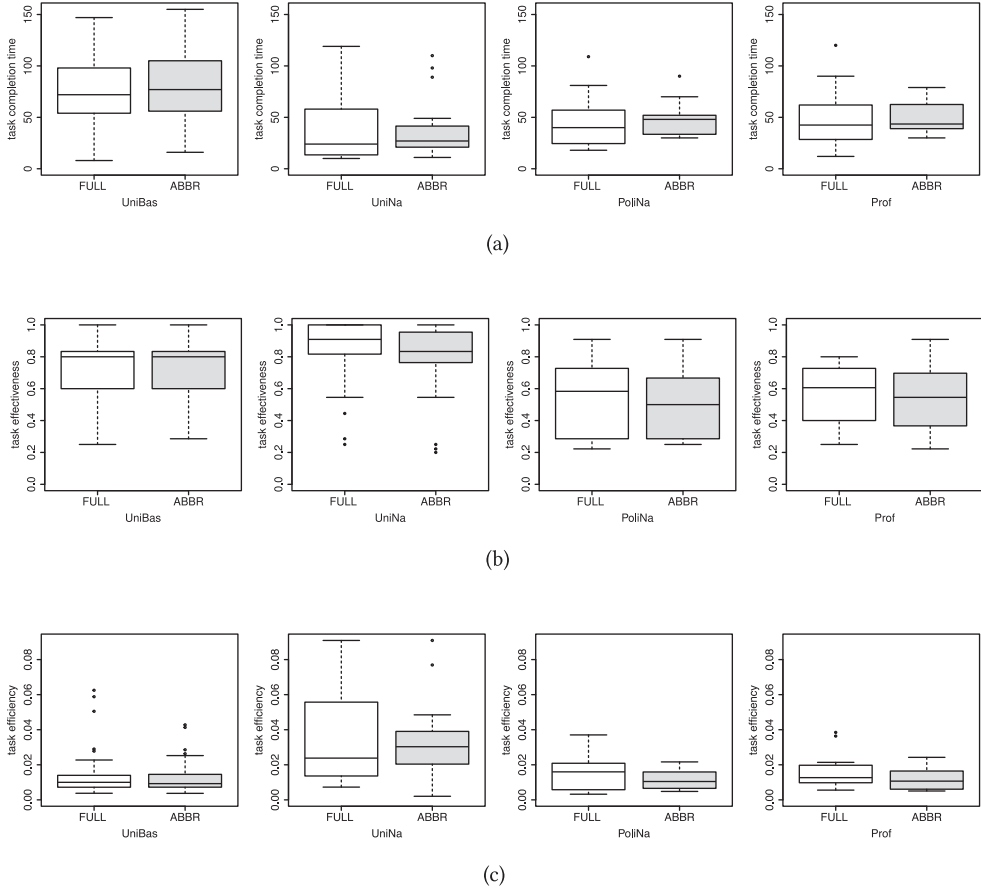


Fig. 4. Boxplots for task completion time (a), task effectiveness (b), and task efficiency (c) grouped by method and experiment.

Table 10. Results for Hn0, Hn1, and Hn2

Hypothesis	Method	Experiment	Trial	Object	Method:Experiment	Method:Trial	Method:Object
Hn0	0.26	< 0.001	< 0.001	0.939	0.993	0.791	0.734
Hn1	0.538	< 0.001	< 0.001	0.95	0.848	0.345	0.881
Hn2	0.103	< 0.001	0.093	0.986	0.9	0.333	0.803

values for these variables is graphically summarized by the boxplots in Figure 4. For example, we report the boxplots for task completion time grouping the values by experiment and method in Figure 4(a). Descriptive statistics and boxplots indicate overall that in all the experiments there is not a huge difference between FULL and ABBR.

4.2 Hypotheses Testing

Results for the null hypotheses Hn0, Hn1, and Hn2 are summarized in Table 10. In particular, we report for each hypothesis the p-values the multivariate linear mixed model returned for: Method,

Table 11. Results for Hn3_X

X (response variable)	Language	Method:Language	Mean improv./reduction
task completion time	0.009	0.895	−26.899%
task effectiveness	<0.001	0.654	−26.465%
task efficiency	0.317	0.554	—

Experiment, Trial, and Object, and the interaction between Method and the other explanatory variables (i.e., Method:Experiment, Method:Trial, and Method:Object).

4.2.1 Hn0: Task Completion Time.. As for task completion time, one of the assumptions to apply a multivariate linear mixed model was not verified because the Shapiro test returned a value less than 0.05 on the residuals. Therefore, we performed an exponential transformation to apply the multivariate linear mixed model on task completion time [76]. The built model did not allow us to reject Hn0 (see second column in Table 10). That is, the effect of Method is not statistically significant for task completion time. The model includes two significant variables, Experiment and Trial. That is, the participants in UniBas spent more time than the participants in the other experiments. As for Trial, the participants spent less time to accomplish tasks in the second trial. We did not observe any interaction between Method and the other explanatory variables and the effect of Object is not significant.

4.2.2 Hn1: Task Effectiveness.. Results indicate that there is not a statistically significant difference between ABBR and FULL (p-value = 0.538). As shown in Table 10, there is a positive effect of Experiment and Trial on task effectiveness. The participants in UniBas and UniNa achieved better task effectiveness values than the participants in the other two experiments. As for trial, we observed that the participants achieved on average better task effectiveness values in the first trial. We can postulate that participants got tired when passing from the first to the second trial. It is worth mentioning that we did not observe any interaction between Method and the other considered explanatory variables and the effect of Object is not significant.

4.2.3 Hn2: Task Efficiency.. One of the assumptions to apply a multivariate linear mixed model was not verified for task efficiency (the Shapiro test returned a value less than 0.05 on the residuals). This was why we applied a logarithmic transformation. On the basis of the results shown in Table 10, we did not reject Hn2, i.e., the effect of Method is not statistically significant on task efficiency because the p-value is larger than 0.05 (i.e., 0.103). The results of the multivariate linear mixed model indicate a positive effect of Experiment, namely, the participants in UniNa obtained better task efficiency values. We observed neither a significant effect of the other explanatory variables nor a significant effect of the interaction between Method and these explanatory variables.

4.2.4 Hn3_X: C vs. Java.. In Table 11, we report the p-values of a multivariate linear mixed model used to study if the programming language with which the experimental objects have been implemented (i.e., the explanatory variable Language) affects results. We had to apply a logarithmic transformation (for each response variable) since residuals did not follow a normal distribution (i.e., the Shapiro test returned a value less than 0.05 on the residuals). The results of our statistical analysis show the presence of a statistically significant difference for the response variables: task completion time and task effectiveness. That is, the built models allowed us to reject the null hypothesis Hn3 on these two dependent variables. We can also observe that the values for the mean

Table 12. Descriptive Statistics of the Participants' Results for the Manual Expansion of the Abbreviated Identifiers

		Min	Max	Mean	St. Dev.
UniBas	P	0.467	1	0.765	0.116
	R	0.037	0.852	0.372	0.172
UniNa	P	0.672	1	0.824	0.103
	R	0.52	0.988	0.761	0.145
PoliNa	P	0.713	1	0.913	0.075
	R	0.24	0.885	0.493	0.201
Prof	P	0.8	1	0.95	0.061
	R	0.08	0.65	0.37	0.212

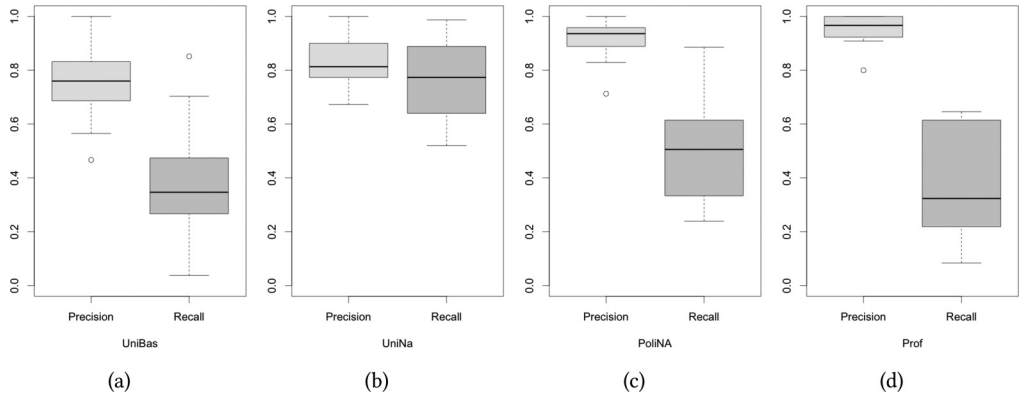


Fig. 5. Boxplots on the results related to the participants' manual expansion of abbreviated identifiers.

percentage reduction¹⁴ suggest that participants spent less time (-26.899%) and obtained worse task effectiveness values on the applications implemented in Java (-26.465%). We also verified the interaction between Method and Language (i.e., Method:Language). Results suggest the absence of such kind of interaction.

4.3 Expanding Identifiers

We estimated correctness and completeness of the identifier expansion by the precision (i.e., $P = \frac{\#(\text{identifiers correctly expanded})}{\#(\text{unique identifiers expanded})}$) and recall ($R = \frac{\#(\text{identifiers correctly expanded})}{\#(\text{unique identifiers present})}$) measures, respectively. It is worth mentioning that an identifier was considered correctly expanded if and only if all its compounding abbreviated words were expanded correctly.

Descriptive statistics (minimum, maximum, mean, and standard deviation values) on precision and recall are reported in Table 12, while we show the boxplots for these measures in Figure 5. We can observe that correctness is higher than the completeness. The shown pattern holds in all the experiments. Indeed, in UniNa the difference between correctness and completeness is not so high and the values of the measures to estimate correctness and completeness are very close to 1 (see Figure 5). We will better discuss the observed pattern in Section 7.1.

¹⁴It can be considered a more intuitive effect size indicator.

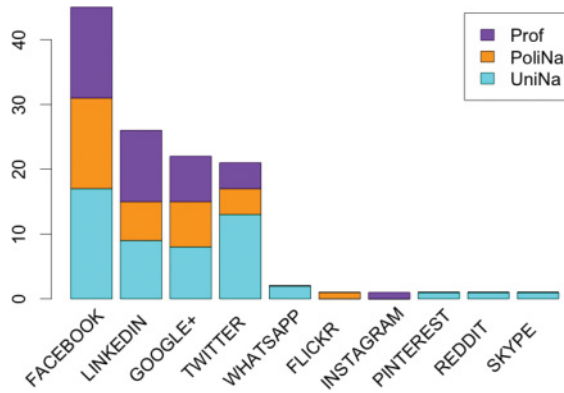


Fig. 6. Responses to question Q8.

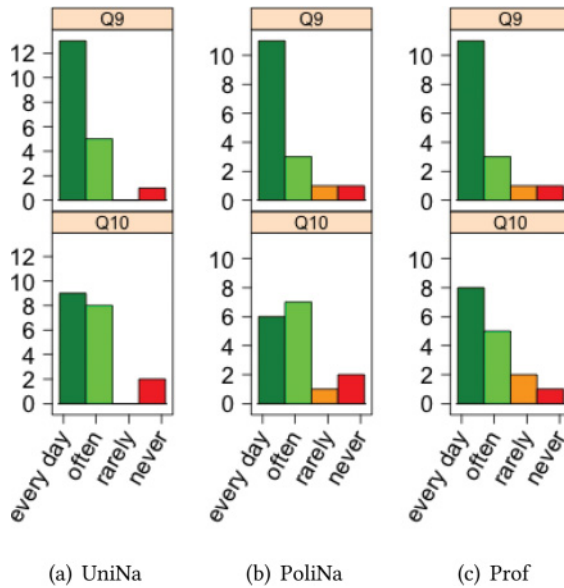


Fig. 7. Responses to questions Q9 and Q10: UniNa (a), PoliNa (b), and Prof (c).

4.4 Post-Experiment Survey Questionnaire

We first summarize results from responses to post-experiment survey questionnaires of all the experiments. Successively, we present results on the responses to the statements from Q8 to Q12 for UniNa, PoliNa, and Prof. The responses to Q8 are graphically summarized in Figure 6, while those to Q9 and Q10 are summarized in Figure 7.

Time needed to carry out the experiments was considered appropriate in all experiments. For each experiment, the greater part of participants' responses were *strongly agree* or *agree* for Q1 (enough time to perform tasks). As for Q2, a similar trend was observed. In particular, participants' responses were *strongly agree* or *agree* on the clarity of the task objectives. Participants also found clear the bug reports we gave them to find and fix faults (Q3). The greater part of participants' responses to Q3 was either *strongly agree* or *agree*. Participants to UniBas were those that found

it more difficult to understand abbreviated identifiers. In general, there were no unanimity in the responses given on Q4 in all the experiments. As for Q5, all participants in UniBas, UniNa, and PoliNa agreed that the presence of abbreviated identifiers made fault fixing difficult. Participants in Prof were more neutral on this point. Participants in PoliNa and UniNa disagreed that there was no difference in using abbreviated or full-word identifiers while fixing faults in source code (Q6). It seems that the presence of abbreviated identifiers is more problematic in the case of C code and for less experienced participants. Participants found the experiments useful (Q7).

Responses to Q8 (see Figure 6) indicate that the most used social media were Facebook (45 out of 51), LinkedIn (26 out of 51), Google+ (21 out of 51), and Twitter (21 out of 51). The greater part of participants used a social medium every day and exploit social media slang to communicate (Q9). The greater part of participants in UniNa, PoliNa, and Prof used every day at least one social medium. As for the use of social media slang, a slightly different pattern was present in the responses to Q10. That is, slang was used every day or often in most cases (see on the bottom of Figure 7).

As for responses to Q11 and Q12, participants in UniNa, PoliNa, and Prof believed that the use of social media slang was changing how developers wrote identifier names and source code comment. Indeed, the participants in these experiments believed that the use of social media was affecting slightly more how identifiers were written. It is worth mentioning that mostly all participants that asserted to use social media slang also positively answered Q11 and Q12.

5 ETHNOGRAPHICALLY INFORMED STUDY

We performed an ethnographically informed study to better interpret results from our family of experiments. We studied the developer's behavior, while they had to fix actual faults in an open source application implemented in Java. We were also interested to investigate the possible differences between the developers' behavior when they were provided with source code with either abbreviated or full-word identifiers. To address these goals, we adopted a design similar to that used in the experiments of our family. This design also presents some similarities with that used in the exploratory study by Robillard et al. [54]. Similar to this latter study, our ethnographically informed study relies on the following two main assumptions:

- There is a relation between program investigation behavior and the success (or not) with which developers can perform fault fixing tasks.
- Personal characteristics of a developer (e.g., the skill, experience, and expertise) are reflected in his/her program investigation behavior.

Rejecting the first assumption would imply that a developer could investigate only irrelevant source code and still correctly fix a fault. Rejecting the second assumption would imply that the program investigation behavior of a developer is either completely algorithmic or completely random. Our two assumptions allowed us to focus on the characteristics of developers' investigation behavior that influence participant's effectiveness without having to consider the factors explaining the program investigation behavior. We believe that our results are then more useful and generalizable.

According to our assumptions, the adopted experimental methodology met two specific requirements: realism and reproduction. To pursue realism, we needed to study a situation representative of realistic fault fixing tasks. In practice, this meant a real application (large enough and developed by multiple people over multiple evolution cycles) with real faults to be fixed so as to require participants a significant amount of effort to investigate the application and to fix these faults. To increase realism participants in the study were junior Java professional developers. As for the reproduction requirement, we needed to be able to contrast the behavior of successful and

unsuccessful reproduction. The rationale here is to identify the characteristics of program investigation behavior associated with successful fault fixing tasks. This requirement implies some form of reproduction of the task. Additionally, because success can be heavily influenced by the kind of task, programming language, and other factors independent of developers, we needed to control the fault fixing task and the environment in which it was performed. These requirements are conflicting because, as we studied realistic tasks, we faced increasing problems controlling for the factors that might influence results, and increasing difficulty analyzing enough of the data and information collected to account for the complexity of the observed phenomenon. In some way, these requirements (realism and reproduction) are conflicting one another; we needed to study realistic tasks in a realistic setting and to face increasing problems controlling for factors that might influence the results due to the complexity of the phenomenon to observe. To deal with these issues, we performed an ethnographically informed study with six different professional developers. This offered a reasonable tradeoff between the cost and detailed qualitative analysis and the generalizability of the results. In the following of this section, we provide details on the design (e.g., context, experimental setting, threats to validity, and so on) of this study.

5.1 Definition and Context

Prior to fixing a fault, developers must inevitably investigate the source code of the target application to find and understand source code affected by the fault fixing. The way a developer investigates a source code with abbreviated identifiers can yield important insights into fault fixing. Therefore, we were interested in exploring the following topics in our ethnographically informed study:

- *how* young professional developers investigate unfamiliar Java source code, without comment and with abbreviated identifiers, when fixing actual faults;
- *why* do abbreviated identifiers (with respect to full-word ones) penalize (or not) fault fixing?
- *what are the characteristics* the developers believe source code and identifiers, in particular, must have so that they can successfully identify and fix faults.

We invited (by e-mail) 11 Java professional developers in our ethnographically informed study. These developers were employees of software companies in the industrial contact network of the software engineering research group at the University of Basilicata. They worked on or participated in research projects conducted by this group. 6 out of 11 developers accepted our invitation. These developers had from 1 to 3 years of work experience as professional Java programmers. They worked in six small/medium companies located in southern Italy. Regarding the industrial domains, one company worked in the area of software consultancy and two companies developed web-based systems and apps for smart devices as their main business activity. Two professionals had a Master's degree in Computer Science and Engineering, while four had a Bachelor's degree in Computer Science. All the professionals took their degree from the University of Basilicata. In their academic carrier, they passed (basic and advanced) Java programming exams. The participants had experience with the execution of maintenance and evolution tasks performed on unfamiliar source code. They also had knowledge of testing approaches and techniques (e.g., unit testing, integration testing, and system testing). The participation in the study was on a voluntary basis, i.e., we did not pay professional developers. These developers participated in the study outside their work hours. This choice allowed us to have motivated participants. They did not know the goals of our study. The target application for our study was JGraphT.¹⁵ It is a free Java graph library providing mathematical graph-theory objects and algorithms. The participants declared to be familiar with the

¹⁵<http://jgraph.org>, version 0.7.0.

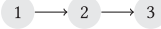
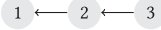


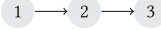

ID	1862338
Title	Edges not removed in reverse graphs
Description	<p>After creating the reverse graph from a directed multigraph, when removing the edges from a given vertex to another one in the reverse graph, these edges are not removed. For example, consider the following directed multigraph (there are two edges from 2 to 3):</p>  <p>Its reverse graph (there are two edges from 3 to 2) is:</p>  <p>When removing the edges from 3 to 2 in the reverse graph, then the obtained graph should be:</p>  <p>Actually, the obtained graph is:</p>  <p>Namely, the edges from 3 to 2 are not removed.</p>
ID	892621
Title	Listenable graphs do not fire some events regarding the removing of edges
Description	<p>When removing a vertex from any kind of graph, the incoming and outgoing edges of this vertex are removed too. Thus, if a listenable graph has been created from another graph and a vertex has been removed from this listenable graph, then the listenable graph should fire events regarding the removing of the vertex and its incoming and outgoing edges (if they exist). For example, consider the following listenable graph:</p>  <p>When removing the vertex 2 from the listenable graph, then the edges from 1 to 2 and from 2 to 3 are removed too. The obtained listenable graph is:</p>  <p>The listenable graph should fire 3 events: (i) removing of the vertex 2, (ii) removing of the edge from 1 to 2, (iii) removing of the edge from 2 to 3. Actually, only the first event is fired whereas the other two are not fired.</p>

Fig. 8. Used bug reports for JGraphT.

problem domain of JGraphT because they passed exams on graph theory and on algorithms and data structures in their academic career. Participants were not familiar with the codebase used in the study.

Some descriptive statistics of this application follow: 6161 LOC, 166 types, and having 19.5% of the identifiers abbreviated. As for LOC, we considered source code without comments. Reported source code metrics have been gathered by the Understand tool. The percentage of abbreviated identifiers is computed with respect to the number of full-word identifiers in the original version of JGraphT. The number of abbreviated identifiers was manually counted and the high value of abbreviated identifiers confirms that developers widely use abbreviated identifiers. We used JGraphT because its faults are well documented in its publicly available software repository. This application is also used in a number of empirical studies (e.g., [48, 75]). From the JGraphT software repository, we randomly selected two real faults so that their fixing fitted participants' time constraints; they gave their availability for a fixed amount of time, namely, 3/4 hours. In Figure 8, we show the bug reports used in the study. Similar to the experiments in our family, a bug report contained an ID (the identifier the original developers used) and a title (a short description) and a long description of the failure (i.e., the difference between expected and observed behavior). No reference to the source code was present in these reports to avoid affecting the identification and the fixing of the faults.

The fourth author of this article produced two versions of JGraphT adopting the same procedure described in Section 3.2. One version contained only abbreviated identifiers, while the other only full-word identifiers. The source code layout was the same in both these versions. Source code comments and test cases were omitted from each version.

5.2 The Setting

For the scope of this ethnography, we kept the experimental setting as close as possible to the natural environment in which the developers carry on their everyday work activities. Describing the setting is a good practice in ethnographically informed studies, as the spatial organization could be relevant insofar as developers working to accomplish a given task [69].

The participants worked on the two faults following a fixed schedule. The participants and the observer (the fourth author) established this schedule before the study. Only the observer and a participant were present in each experimental session. All the participants used the same laptop to carry out the task. These measures were taken to minimize any possible bias arising from the differences in physical experimental settings. All the participants were familiar with Eclipse, namely, the IDE used in the study. They used Eclipse in their work activities.

5.3 The Study

The study was conducted by a single observer (the fourth author) from November to December 2016 and it was founded on one-to-one sessions between the observer and each participant. This experimental procedure is customary in ethnographically informed studies (e.g., [57, 58]). We conducted our study in Italian to avoid biasing the study results since participants were Italian professionals and they may have different familiarity with the spoken English language.

The observer (if needed) engaged with the participants (without conditioning their work habit) focusing on both the application and solution domains of JGraphT. It was important for the observer to participate in the study because (i) the ethnographic approach encourages the participation of the observer to the study [70]; (ii) the observer could appreciate the perspective of the developers while carrying out the assigned tasks; and (iii) the observer could get information on how faults were identified and fixed. The observer avoided influencing the participants during the execution of the fault fixing tasks. To mitigate subjective assumptions, the observer had to consider all the activities related to our study as “strange,” as suggested by Sharp et al. [70]. The data and the information regarding these tasks were collected by using contemporaneous field notes, audio recordings of discussions, copies of various artifacts (e.g., source code), and screen recordings. We pseudo-randomly divided the participants into two groups. This was done when the list of the participants was closed, i.e., at the end of the time slot we fixed to receive answers by e-mail from the invited developers. The fixed time slot was 1 week after the invitation sent to the 11 developers we asked to participate in the study. The assignment to these groups is considered pseudo-random because we imposed that each group had to contain the same number of participants. A group of participants worked on the version of JGraphT with full-word identifiers, while the other on the version with abbreviated identifiers. The behavior of the developers in the group that worked on the JGraphT version with full-word identifiers represents a sort of baseline for comparison.

5.4 Design

The study was divided into four phases. To minimize potential investigator bias, each phase was described entirely through written instructions. In any phase, the participants could ask for clarifications, but we established guidelines restricting answers from the observer to clarification on the study and its material.

- *Pre-study task.* To obtain participants’ demographic information, we asked them some questions (e.g., years of work experience and preferred IDE). To be sure that all the developers had sufficient familiarity with basic notions on the graph data structure, we asked them to read through a tutorial, but they could end the training period at their discretion.

Before continuing on to the next phase of the study, the participants had to answer a few proficiency questions (e.g., *what is a simple graph?*) on the contents covered in this tutorial.

- *Program Investigation.* We asked participants to read some preparatory material about JGraphT. This material included an excerpt from the user's documentation of JGraphT and instructions on how to launch it through the official demo programs present in the JGraphT distribution. The participants were allowed to execute demo programs at any point during the study. No demo program revealed the two faults considered in our study.

The participants were given 30 minutes to investigate the code in preparation for the actual fault fixing task to be performed in the subsequent study phase (i.e., Fault Identification and Fixing). The participants were informed about the possibility of taking notes in a text file, but we did not explicitly instruct them on how to do that. We allowed the participants to execute JGraphT (using or not demo programs), but they could not modify the code. Our intent in this phase was to introduce some control over the study, and to prevent the case where a professional would try to perform fault fixing with almost no prior investigation on the source code of the target application. Although this is possible in the case of trivial tasks, we knew from designing our qualitative study that the possibility of succeeding at the task without investigating the program was extremely unlikely [54]. We discuss the potential impact of this choice on the results in Section 5.5.

- *Fault Identification and Fixing.* In this phase, the observer asked the participants to fix the faults one at a time. He gave a developer one bug report first and then the other (see Figure 8). Differently from the experiments in our family, we did not ask participants to indicate the portion of modified source code because the observer participated in the study. The participants were given 3 hours to complete the Fault Identification and Fixing phase. This was the needed time we estimated to fix both the faults.
- *Post-Experiment Task.* We asked participants to fill in the post-experiment survey questionnaire shown in Table 7, which is the same questionnaire as we asked the participants in UniNa, PoliNa, and Prof to fill in. Since the participants might have a different notion of social-media slang, the observer provided them with a definition of social-media slang and some well known terms of such slang.

The observer could provide support for the phases before. He could clarify concerns related to the study and/or to the questions of the used pre- and post-questionnaires. On the other hand, the observer immersed himself and participated (if needed) in all the phases introduced just before, but he did not disturb or change the natural setting of the study.

5.5 Threats to Validity

5.5.1 Internal Validity. The experimental setup may have affected the behavior of the participants. In particular, the separation of a fault fixing task into the investigation and fault fixing phases may have caused adjustments to the participants behavior.

5.5.2 External Validity. The number of participants could limit the generalizability of our findings. This is not a major issue here because we used our qualitative study as a complementary investigation to deepen results from a family of four controlled experiments conducted with a total of 100 participants. Furthermore, results could be also affected by the software development environment and programming language. Although the participants were familiar with Eclipse and Java, this additional factor limits the generalizability of the study to similar conditions. Another threat to external validity could concern the application used in the study.

5.5.3 Construct Validity. Social factors should be taken into account when evaluating qualitative and quantitative findings (e.g., evaluation apprehension). To deal with this kind of threat, the observer used an informal approach to interact with the participants in the study. Participation in the study was on a voluntary basis. Voluntary participants are generally well motivated and then results could be unintentionally affected. Finally, the influence that the observer could practice on the professionals may also affect the validity of the results.

6 RESULTS FROM THE ETHNOGRAPHICALLY INFORMED STUDY

We present the findings from our ethnographic analysis following a standard approach (e.g., [9]). We first present the main themes emerging from our collected information and data and then the results from the post-experiment task. We conclude highlighting some quantitative results from our ethnographically informed study.

6.1 Ethnographic Analysis

The goal of our ethnographic analysis was to find insights from recurrent themes in the developers behavior, while conducting fault fixing. The meaning behind the observed behavior must be inferred from the details of the collected information and data [69]. Our analysis is based on the following steps:

Reflecting: The observer first reflected upon the experience gained in his immersion and then used the data and the information to recollect, revisit, and reconsider what was found from such an experience.

Discussing: The observer and the other authors discussed the audio and video recordings, the source code written by the participants, and other artifacts such as the data collected in the questionnaires.

Confirming themes: When a theme appeared to be emerging in the behavior of a developer working on source code with abbreviated identifiers, we searched for data and information in the same group of developers to contradict this theme. If no contradictory evidence emerged, then the theme was confirmed, otherwise it is not confirmed.

Verifying confirmed themes: When a theme was confirmed in the step before, we searched for data and information in the group of developers working on source code with full-word identifiers that could contradict this theme. If no contradictory evidence emerged, then the theme holds in both the groups of developers. If contradictory evidence emerged, we were able to find a difference in the behavior of the two groups.

This kind of analysis proceeded iteratively. This approach required a considerable degree of effort, especially in the validation of potential themes with respect to the collected data and information. In our ethnographic analysis, we found several candidate themes about the developers' investigation behavior of both the successful and unsuccessful fault fixing. Among these themes, we confirmed the following ones:

- (1) Identifier Expansion;
- (2) Dealing with Focus Points;¹⁶
- (3) Methodical¹⁷ Fault Fixing;
- (4) Text-Based Search;

¹⁶A focus point is a point in the source code that is relevant to a given task [73].

¹⁷As done by Robillard et al. [54], we use the term methodical in its general sense to indicate a developer's behavior characterized by method and order.

- (5) Inattention Blindness;¹⁸
- (6) Code Recommenders;
- (7) Characteristics of Identifiers.

From the confirmed themes, we derived results about the intrinsic factors influencing the success (or not) of developers that perform fault fixing tasks. We formulate these results as a high-level statement. In the following subsections, we illustrate and detail each confirmed theme.

6.1.1 Identifier Expansion.. The following statement summarizes the main result from this theme.

Developers try expanding identifiers only when it is needed to better investigate source code to fix a given fault.

When developers deal with unfamiliar source code they build a sort of mental model [41, 73] of that code. We observed that the participants in our study also built a mental model of the code when investigating it to fix faults. They used non-lexical information (e.g., syntactical structure of the code and/or its control and dataflow) without paying attention to the abbreviated identifier names. They tried expanding abbreviated identifiers only when they were not able to build their mental model. In other words, in a few cases the developers also needed the lexical information of abbreviated identifiers to enrich their mental model.

6.1.2 Dealing with Focus Points.. We summarize the result from this theme as follows:

Developers need to extend their focus point to expand abbreviated identifiers.

There could be some abbreviated identifiers that the developers were not able to easily expand in their mind. That is, they were not familiar with the abbreviation of one or more abbreviated compounding words of an abbreviated identifier. If developers believed that this abbreviated identifier was useful to build their mental model, then they expanded the focus point to infer the meaning of the unfamiliar abbreviated compounding words. The developers used the following strategies to expand their focus point: (i) exploring software static and syntactic relationships starting from the abbreviated identifier; (ii) exploring dynamic relationships concerned the abbreviated identifier by debugging the application; (iii) exploring lexically and/or semantically relationships of unknown compounding words of the abbreviated identifier on the basis of known abbreviated identifiers, e.g., if a developer understands that g means graph, then he/she can infer that e and v stand for edge and vertex, respectively; and (iv) combining the preceding strategies.

6.1.3 Methodical Fault Fixing.. The statement that summarizes the result from this theme is

To fix faults in the source code with abbreviated identifiers developers apply a more methodical approach as compared with developers working on source code with full-word identifiers.

Developers that fixed faults in source code with abbreviated identifiers (with respect to those provided with source code having full-word identifiers) investigated enough of the JGraphT code to understand its high-level structures. It seemed that these developers prepared a detailed plan of the identification of faults and of the changes to be made on the source code to fix these faults. In contrast, the behavior of developers dealt with full-word identifiers seemed more opportunistic (antonym of methodical) because they relied heavily on code skimming and guessing. The methodical approach did not lead to a longer time to finish fault fixing tasks.

¹⁸It is the failure to notice a fully visible, but unexpected object because attention was engaged on another task, event, or object [45].

6.1.4 *Text-Based Search..* We summarize the result from this theme as follows:

Abbreviated identifiers make it difficult to perform text-based searches in source code.

To help developers to discover and understand parts of a program, Eclipse (and also other IDEs) provides features to perform text-based searches on the source code by using queries. We noted that all the developers (i.e., professionals provided with source code having either abbreviated or full-word identifiers) used as textual queries both concepts of the problem and solution domain and identifier names. Developers provided with source code having abbreviated identifiers were penalized in two directions. First, they chose as text-based queries the concepts in their full-word variant because they were not aware that identifiers were all abbreviated in the source code. This finding is coherent with that Sillito et al. [73] observed in their study: developers search for types representing domain concepts or for entities named something like concepts. Second, there could be several ways to abbreviate an identifier and the likelihood to choose the wrong textual query is high and could affect search results.

6.1.5 *Inattention Blindness..* We can summarize the result from this theme as follows:

Inattention blindness is slightly more frequent when source code contains abbreviated identifiers.

We noted that inattention blindness arose in the two groups of developers. However, this phenomenon seemed more frequent in the group of developers we asked to perform fault fixing in the source code with abbreviated identifiers. In particular, two developers in the group with abbreviated identifiers underwent this phenomenon and one developer in the other group. Independently from the group, it happened that developers did not acknowledge source code fragments that were relevant to reproduce, identify, or fix a given fault, when these fragments were showed accidentally. Inattention blindness is not refuted by the successful ones to fix faults since all the three participants (completely or partially) fixed the fault. This finding is coherent with that by Robillard et al. [54] observed in their study.

6.1.6 *Code Recommenders..* This theme produced the following result.

Developers use Eclipse code recommenders (also known as intelligent code completion) to reduce the navigation among software entities while identifying and fixing faults in source code.

We noted that developers in both the groups used the Eclipse code recommenders to identify and fix faults. Indeed, developers used this feature to obtain recommendations of likely entities to call/use by triggering code completion on another entity (e.g., while writing the name of a method, the tool suggests possible completions, namely, the compatible methods). Code recommenders automatically analyze existing code and extract common usage rules and patterns from it. The developers seemed to be aware of the power of this feature and used it to reduce navigation among software entities while they performed fault localization and fixing. As elaborated for the Text-based Search theme, developers could be slightly penalized if they worked on abbreviated identifiers; there could be several ways to abbreviate an identifier and the likelihood to choose the wrong prefix (from which triggering code completion) is higher and this could affect recommendations.

6.1.7 *Characteristics of Identifiers..* We can summarize the result from this theme as follows.

Abbreviated identifiers are not a major obstacle to fault identification and fixing.

This theme emerged from the developers in the group provided with the version of JGraphT with abbreviated identifiers. In particular, two out of three developers did not perceive abbreviated identifiers as a major issue to the fault identification and fixing even if (as shown before) they

Table 13. Responses to the Questions from Q8 to Q12

Group	Participant	Questions				
		Q8	Q9	Q10	Q11	Q12
Full-word Identifiers	P1	WhatsApp	Every day	Rarely	No	No
	P2	Telegram	Every day	Rarely	No	No
	P3	Facebook	Every day	Every day	Yes	Yes
Abbreviated Identifiers	P4	Facebook	Every day	Never	No	No
	P5	Telegram	Every day	Rarely	No	No
	P6	Telegram	Every day	Often	No	No

accomplished the fault fixing task in a way slightly different from the developers in the group with full-word identifiers. Both these developers were able to find and successfully fix one fault and the second was partially fixed (see Section 6.3). The ability to fix faults of the participants provided with source code with abbreviated identifiers was not inferior to that of the participants provided with source code having full-word identifiers.

6.2 Post-Experiment Task

We report the responses to the questions of the post-experiment survey questionnaire in Table 13. The responses to Q8 indicate that Telegram is the most used social medium, while responses to Q9 suggest that all the participants used social media every day. As for Q10, most of the participants stated that they rarely used social media slang. Finally, the responses to Q11 and Q12 show that participants did not believe that social media slang was changing how developers choose identifiers and write source code comments. Summarizing the participants in the qualitative investigation gives a perspective different from that of the participants in the family of experiments (i.e., UniNa, PoliNa, and Prof).

6.3 A Quantitative Perspective on the Study

We evaluated the source code the participants wrote during the fault fixing task. The fourth author of this article inspected the patches the participants wrote to fix the faults in JGraphT. To characterize the success level of each fixed fault, we used the following classification that is inspired by the classification proposed in [54]:

- **Success.** The participant completely fixed the fault.
- **Buggy.** The participant partially fixed the fault. That is, he/she wrote some source code that worked in most cases.
- **Unworkable.** The participant did not fix the fault.
- **Not-attempted.** The participant did not make an attempt to fix the fault.

In Table 14, we report the characterization of the success level for each assigned fault (i.e., 1862338 and 892621) and participant. All the participants provided with source code with full-word identifiers successfully fixed the fault 1862338, while two out of three participants working on source code with abbreviated identifiers successfully fixed this fault. Nobody successfully fixed the fault 892621. One participant working on source code with full-word identifiers partially fixed the fault 892621. On the other hand, two participants of the group provided with source code with abbreviated identifiers partially fixed the fault 892621. The participant P5 spent all her time to fix the fault 1862338. This was why she did not try to fix the fault 892621. It seems that the fault 892621

Table 14. The Characterization of the Success Level for Each Subtask (of the Fault Fixing Task) and Each Participant

Group	Participant	Faults	
		1862338	892621
Full-word Identifiers	P1	SUCCESS	BUGGY
	P2	SUCCESS	UNWORKABLE
	P3	SUCCESS	UNWORKABLE
Abbreviated Identifiers	P4	SUCCESS	BUGGY
	P5	BUGGY	NOT-ATTEMPTED
	P6	SUCCESS	BUGGY

was more difficult to find and fix than the fault 1862338. Our analysis on the success level does not show a meaningful pattern. That is, it seems that the kind of identifier does not affect fault fixing, so confirming results from our family of controlled experiments.

7 OVERALL DISCUSSION

In the following subsections, we discuss the links between observed results and defined research questions. We reinforce this link with the results from our ethnographically informed study when possible. We conclude delineating practical implications from our qualitative and quantitative results.

7.1 Linking Results and Research Questions

As for RQ1 (i.e., does the presence of either abbreviated or full-word identifiers in source code penalize the effort to accomplish fault fixing tasks in that code and the effectiveness and efficiency to accomplish this kind of task?), we were not able to reject the null hypotheses Hn0, Hn1, and Hn2. From the qualitative part of our investigation, we found results that are coherent with those from our family of experiments and that provide a qualitative explanation for them. In particular, professional developers in our ethnographically informed study did not perceive abbreviated identifiers as a major obstacle to fault identification and fixing. The lack of a significant difference between fixing faults in source code with either abbreviated or full-word identifiers could be justified because developers apply a more methodical approach when fixing faults in source code with abbreviated identifiers. In some way, the use of a more methodical approach allows developers to alleviate possible perils in dealing with abbreviated identifiers (e.g., text-based search and inattention blindness). Concluding, on the basis of our outcomes and considerations it seems that we cannot positively answer RQ1.

As for RQ2 (i.e., does programming language affect the effort, effectiveness, and efficiency to execute fault fixing tasks?), the data analysis allowed us to reject Hn3 on task completion time and task effectiveness. In other words, it seems that fault fixing on C source code is faster and the participants are more effective in executing this task independently from the kind of identifiers. It is important to note that the interaction between method and programming language was not significant on all three response variables. Therefore, a possible justification for these outcomes could be related to the inherent differences between C and Java, namely, the programming paradigm: C is procedural, while Java is object-oriented. For example, the information needed to comprehend C code and then to identify and fix faults is spread in a lower number of files. This could reduce the effort needed to fix faults in source code. We can also postulate that the lower the number of files, the better task effectiveness is. These postulations are supported by the results

by Ko et al. [40]. In particular, the authors observed that developers deal with existing source code searching for relevant pieces of code by following incoming and outgoing dependencies of relevant code. A more recent study found similar results [60]. It could also be possible that the differences shown before could be related to the IDEs rather than the programming language. The used IDEs (Dev-C++ in UniBas and UniNA and Eclipse in PoliNa and Prof) represent a threat to the conclusion validity (see reliability of treatment implementation in Section 3.10.4). However, from our qualitative study we observed that professional developers largely use Eclipse features when investigating source code. Many of these features (e.g., intelligent code completion) are not present in Dev-C++ and seemed appreciated and largely used when fixing faults. On the basis of both these considerations and the obtained results, we can positively answer RQ2. However, we believe that caution is needed and future work is advisable.

On the basis of our results, we are not able to definitively answer RQ3 (i.e., can developers correctly and completely expand abbreviated identifiers?) even if it seems that participants were able to correctly expand the abbreviated identifiers present in source code on which they performed fault fixing tasks. Participants did not have any particular difficulty in associating the right meaning to the abbreviated compounding words of identifiers. In fact, the obtained precision values are high in all our experiments (i.e., the mean values of precision range in between 0.765 and 0.95). In other words, it seems that the programming language and the kind of participant do not affect the values for the P measure. The participants often expanded a part of the list of the abbreviated identifier names with respect to the whole set of identifiers we asked them to expand. This aspect affected the R values in all the experiments that are not so high as compared with the P values (see Figure 5). This is true in all the experiments with the only exception of UniNa (0.761). As far as these outcomes, we can speculate that (i) participants did not encounter all the abbreviated identifiers when dealing with the fault fixing task; (ii) participants expanded identifiers only in case they were confident that an expansion was correct; and (iii) participants did not need to understand the meaning of all the abbreviated identifiers. Results from our ethnographically informed study (see the Identifier Expansion and Dealing with Focus Points themes) and those by Weiser [77] (e.g., he observed that developers focused only on some source code pieces, while debugging unfamiliar programs) provide supporting evidence. Finally, if we compare UniNa and PoliNa (the two experiments with participants with similar development experience), it seems that the programming language could make a difference with respect to R , namely, participants in UniNa obtained better recall values. However, it was impossible to answer on this point analytically because Java and C programs were different from one another and did not include the same identifiers.

As for RQ4 (i.e., do developers perceive fault fixing on source code containing abbreviated identifiers as more difficult/simple than fault fixing on source code containing full-word identifiers?), gathered and analyzed responses from the statements of our post-experiment survey questionnaire suggest that participants in our family of controlled experiments perceived the presence of abbreviated identifiers (with respect to full-word ones) in the source code as problematic while fixing faults. This is a case in which controlled experiments provide insight into the differences between the perceived and the effective disadvantages (or advantages) of administering a given treatment to a participant. It is worth mentioning that the qualitative findings from the ethnographically informed study contrast the results from our family of controlled experiments. Professional developers did not find abbreviated identifiers as a major obstacle to fault identification and fixing. However, we observed some differences between professional developers provided with source code with abbreviated and full-word identifiers. It seems that developers have to adapt their habits to investigate source code when it contains only abbreviated identifiers. We can speculate that fault fixing is not perceived as a major issue when fault fixing is performed on real applications with real faults and developers have a given professional experience. To conclude our study does not

provide conclusive results on how developers perceive abbreviated and full-word identifiers when performing fault fixing tasks even if our outcomes pose the basis for future research on this matter.

7.2 Implications

We delineate here main practical implications for our investigation from both *practitioner/consultant* (simply *practitioner* in the following) and *researcher* perspectives. These implications can be summarized as follows:

- Overall results suggest that the ability of participants to fix faults in unfamiliar source code seems independent from whether identifiers are written in abbreviated form or not. This outcome is relevant because developers could program without paying attention to the identifier form. This implication is especially important from the practitioner perspective. On the other hand, the researcher could be interested in studying how developers deal with identifiers in abbreviated form. The results from the ethnographically informed study pose the basis in this respect. That is, we observed that developers use a different approach to program investigation when working on abbreviated identifiers: the developers adopt a more methodical approach to identify and fix faults extending their focus point and in only a few cases they expand the compounding words of abbreviated identifiers.
- We observed differences in fixing faults in C and Java source code in our family of experiments. In particular, we observed differences in effort and effectiveness constructs. These findings are particularly relevant for the researcher, who could be interested in further investigating how much programming language impacts on task completion time and task effectiveness. Our results produce the basis for future work on this matter.
- The family of controlled experiments is focused on desktop applications implemented in C and Java. The researcher and the practitioner could both be interested in studying whether our results also hold for different kinds of software (e.g., web applications). Finally, it could be of interest for the researcher to study whether our outcomes scale also to applications more complex and larger than those we used in our family of experiments. The presented ethnographically informed study might represent a first step toward this research direction. The results from this study seem to confirm those from our family of controlled experiments on a larger and real application from a different domain with real faults.
- Results suggest that effort and effectiveness constructs seem not to be directly related to one another for the program written in C (UniBas and UniNa). That is, spending on average more time to fix faults does not mean to improve task effectiveness and task efficiency for both FULL and ABBR. This result could be concerned with the participants' experience and it is worthwhile for both the researcher and the practitioner. Our study poses the basis to future investigations in that direction.
- The way people communicate with one another through social media and social media slang is creeping into the today's society. Since source code is very often the only artifact available to comprehend an application and to implicitly communicate with other developers [56, 60], it is easy to imagine that both the communication and the way in which source code is written could change. However, the use of abbreviations is not new in computer programming (e.g., *tmp* and *i* are largely used), but social media slang may make this practice more widespread and extensive in the future. Qualitative results from our family of controlled experiments and from our ethnographically informed study contrast on if social media slang is changing or not how developers write source code. In particular, the participants in UniNa, PoliNa, and Prof believed that social media slang was changing how developers write code. These participants also believed that the use of social media affects

the choice of identifier names more than how source code comments are written. The perspective of the professional developers in the ethnographically informed study was different on the identifier names. The greater part of them asserted that social media slang is not changing how developers choose identifiers. From the gathered data, it seems that the participants who used more often social media and their slang perceived differently from others the change process to how developers choose identifiers. This point is of interest for the researcher, who could want to study if and how the use of social media slang affects computer programming style. In addition, the researcher could be interested in studying if developers' familiarity with social media slang might interact with their habit in choosing and dealing with abbreviations.

8 CONCLUSION

In this article, we present a family of controlled experiments to assess whether abbreviated or full-word identifier names have any effect on the removal of faults in unfamiliar C and Java source code. Our family consisted of four experiments involving a total of 100 participants. Results indicated that the difference in using abbreviated and full-word identifier names is not statistically significant with respect to the effort, the effectiveness, and the efficiency to identify and fix faults. We also conducted a qualitative study on a real application containing real faults. We involved in this study six professional developers with 1–3 years of work experience. Results allowed us to find additional insights and complement those from our family of experiments. We can summarize these insights as follows: fault fixing in source code with abbreviated identifiers needs a more methodical approach and abbreviated identifiers are not perceived as a major obstacle to fix faults in source code.

APPENDIXES

A SEEDED FAULTS IN SOURCE CODE

In this Appendix, we show two examples of faults seeded in AveCalc and Hotel-Reservation. To obtain the shown faulty code, we showed the application of the COR operator to AveCalc and the CFD operator to Hotel-Reservation, respectively. In particular, we report the source code of AveCalc (i.e., the method `setVote`) before having applied the COR mutation operator (at the top of Figure 9). The application of this operator resulted in the source code shown at the bottom of Figure 9. In particular, we replaced an or binary conditional operator (`||`) with an and operator (`&&`). The seeded fault is highlighted in red. The presence of this fault also allows an invalid input, namely, negative integer and values greater than 30. As for Hotel-Reservation, CFD operator was used. We show a chunk of Hotel-Reservation source code before and after fault seeding at the top and at the bottom of Figure 10, respectively. Mutation (i.e., break) is highlighted in red. The presence of this fault breaks out the loop handling only the first element of the array.

```
public void setVote(int vote) {
    if (vote < 0 || vote > 30)
        throw new IllegalArgumentException("Vote must to be a number < 30 and > 0");
    this.vote = vote;
}

public void setVote(int vote) {
    if (vote < 0 && vote > 30)
        throw new IllegalArgumentException("Vote must to be a number < 30 and > 0");
    this.vote = vote;
}
```

Fig. 9. Example of fault seeded in AveCalc.


```

...
clrscr();
printf("R.No. Name NIC Number Check In Check Out\n");
for (iterator = 1; iterator < SIZE; iterator++) {
    if (singleRoom[iterator] == 1) {
        printf("%d %s %s %s %s\n", iterator,
            guest.name[iterator],
            guest.fiscalCodeNumber[iterator],
            guest.checkinDate[iterator],
            guest.checkoutDate[iterator]);
    }
}
...

...
clrscr();
printf("R.No. Name NIC Number Check In Check Out\n");
for (iterator = 1; iterator < SIZE; iterator++) {
    if (singleRoom[iterator] == 1) {
        printf("%d %s %s %s %s\n", iterator,
            guest.name[iterator],
            guest.fiscalCodeNumber[iterator],
            guest.checkinDate[iterator],
            guest.checkoutDate[iterator]);
    }
    break;
}
...

```

Fig. 10. Example of fault seeded in Hotel-Reservation.

Table 15. Descriptive Statistics for Mutation Operator

Operator	ALL		FULL		ABBR	
	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
Conditional Operator Replacement (COR)	0.875	0.336	0.875	0.342	0.875	0.342
Literal Change Operator (LCO)	0.804	0.398	0.888	0.317	0.711	0.455
Variable Replacement Operator (VRO)	0.665	0.473	0.69	0.464	0.642	0.481
Relational Operator Replacement (ROR)	0.625	0.492	0.625	0.5	0.625	0.5
Language Operator Replacement (LOR)	0.585	0.494	0.577	0.496	0.593	0.493
Assignment Operator Replacement (ASR)	0.5	0.503	0.479	0.505	0.521	0.505
Control Flow Disrupt operator (CFD)	0.474	0.5	0.473	0.501	0.475	0.501
Arithmetic Operator Insertion (AOI)	0.422	0.498	0.406	0.499	0.438	0.504
Conditional Operator Insertion (COI)	0.406	0.499	0.562	0.512	0.25	0.447

B ANALYSIS OF SEEDED FAULTS

A seeded fault can be successfully fixed or not. We consider a response variable (i.e., fixed) that assumes 1 if and only if a fault is successfully fixed, while it assumes 0 if a fault is not successfully fixed. In Table 15, we show some descriptive statistics on this variable grouping observation for operator (i.e., kind of seeded fault) and for kind of identifier (FULL and ABBR mean that the source code contained full-word and abbreviated identifiers, respectively). The second and third columns report the mean and the standard deviation for kind of seeded fault and without considering the kinds of identifier. The mean values of the fixed variable assume values in between zero and one. The best value is one, while the worst is zero. One indicates that all the faults seeded with that operator were successfully fixed in the family of experiments. According to the descriptive statistics in Table 15, we can then postulate that it was less difficult for the participants to fix the faults

seeded by applying the COR and LCO mutation operators [36, 37]. The mean values for COR and LCO were 0.875 and 0.804, respectively. There is not a huge difference in the mean values grouping observations for FULL and ABBR. This pattern holds for all the operators with the only exception of COI. It seems that it was less difficult to identify and fix the fault seeded by applying this operator when identifiers were full-word. This difference could be due to the number of observations (16 for FULL and 16 for ABBR). It is worth noting that the standard deviation values are high in many cases because the considered response variable assumes two possible values: 0 and 1.

To complete the study on the kind of operators to seed faults in the application, we also applied a multivariate linear mixed model analysis. We used this kind of analysis to verify the effect of the mutation operator and the kind of identifiers and the presence of a significant interaction between them. Results showed that there is a significant effect of the operator on the fixed variable (p -value < 0.001), while the effect of the identifiers is not statistically significant (p -value = 0.299). That is, some seeded faults were easier or more difficult to fix than others, but this did not depend on the kind of identifier. In fact, the interaction between the kind of mutation operator and the kind of identifier is not statistically significant (p -value = 0.375). According to this insight, we analyzed the faults bearing in mind the kind of operator we applied to seed them. This investigation suggested that those seeded faults that modified the original control and dataflow were more difficult to identify independently from the kind of identifiers. Although this outcome perhaps might be not surprising, the results of this further analysis pose the basis of future investigations on the simplicity/complexity to identify and fix faults seeded by applying mutation operators.

ACKNOWLEDGMENTS

The authors would like to thank people who participated in our family of experiments and in our ethnographically informed study. We would also like to thank Massimiliano Di Penta for his valuable comments and suggestions.

REFERENCES

- [1] Silvia Mara Abrahão, Carmine Gravino, Emilio Insfran Pelozo, Giuseppe Scanniello, and Genoveffa Tortora. 2013. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Transactions on Software Engineering* 39, 3 (2013). DOI: <http://dx.doi.org/10.1109/TSE.2012.27>
- [2] Jorge Aranda, Neil Ernst, Jennifer Horkoff, and Steve Easterbrook. 2007. A framework for empirical evaluation of model comprehensibility. In *Proceedings of Modeling in Software Engineering, ICSE Workshop*. IEEE, 7–13.
- [3] Venera Arnaoudova, Laleh Mousavi Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2014. REPENT: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (2014), 502–532. DOI: <http://dx.doi.org/10.1109/TSE.2014.2312942>
- [4] M. T. Baldassarre, J. Carver, O. Dieste, and N. Juristo. 2014. Replication types: Towards a shared taxonomy. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. ACM, 18:1–18:4.
- [5] V. Basili, F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 25, 4 (1999), 456–473.
- [6] Victor R. Basili and H. Dieter Rombach. 1988. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering* 14, 6 (1988), 758–773.
- [7] K. Beck. 2003. *Test Driven Development: By Example*. Addison Wesley.
- [8] Bethany K. Dumas and Jonathan Lighter. 1978. Is slang a word for linguists? *American Speech* 53, 1 (1978), 5–17.
- [9] Paul Beynon-Davies. 1997. Ethnography and information systems development: Ethnography of, for and within is development. *Information & Software Technology* 39, 8 (1997), 531–540.
- [10] P. Beynon-Davies, D. Tudhope, and H. Mackay. 1999. Information systems prototyping in practice. *Journal of Information Technology* 14, 1 (March 1999), 107–120. DOI: <http://dx.doi.org/10.1080/026839699344782>
- [11] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.

- [12] Gerardo Canfora and Massimiliano Di Penta. 2007. New frontiers of reverse engineering. In *Proceedings of the Workshop on the Future of Software Engineering*. 326–341.
- [13] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. 2003. Issues in using students in empirical studies in software engineering education. In *Proceedings of the International Symposium on Software Metrics*. IEEE CS Press, 239.
- [14] Jeffrey C. Carver, Natalia Juristo Juzgado, Maria Teresa Baldassarre, and Sira Vegas. 2014. Replications of software engineering experiments. *Empirical Software Engineering* 19, 2 (2014), 267–276.
- [15] Marcus Ciolkowski, Dirk Muthig, and Jörg Rech. 2004. Using academic courses for empirical validation of software development processes. *Proceedings of the EUROMICRO Conference* (2004), 354–361. DOI : <http://dx.doi.org/10.1109/EURMIC.2004.1333390>
- [16] Massimo Colosimo, Andrea De Lucia, Giuseppe Scanniello, and Genoveffa Tortora. 2009. Evaluating legacy system migration technologies through empirical studies. *Information & Software Technology* 51, 12 (2009), 433–447.
- [17] Andrew Crabtree, Tom Rodden, Peter Tolmie, and Graham Button. 2009. Ethnography considered harmful. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'09)*. ACM, 879–888. DOI : <http://dx.doi.org/10.1145/1518701.1518835>
- [18] Fabio Q. B. da Silva, Marcos Suassuna, A. César C. França, Alicia M. Grubb, Tatiana B. Gouveia, Cleviton V. F. Monteiro, and Igor Ebrahim dos Santos. 2014. Replication of empirical studies in software engineering research: A systematic mapping study. *Empirical Software Engineering* 19, 3 (2014), 501–557.
- [19] Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. 2005. Towards understanding programs through wear-based filtering. In *Proceedings of the International Symposium on Software Visualization (SoftVis'05)*. ACM, 183–192. DOI : <http://dx.doi.org/10.1145/1056018.1056044>
- [20] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (April 1978), 34–41.
- [21] Eric Enslen, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the International Working Conference on Mining Software Repositories*. 71–80. DOI : <http://dx.doi.org/10.1109/MSR.2009.5069482>
- [22] Gordon Fraser and Andreas Zeller. 2012. Mutation-driven generation of oracles and unit tests. *IEEE Transactions on Software Engineering* 38, 2 (March 2012), 278–292.
- [23] Davide Fucci, Giuseppe Scanniello, Simone Romano, Martin Shepperd, Boyce Sigweni, Fernando Uyaguari Uyaguari, Burak Turhan, Natalia Juristo, and Markku Oivo. 2016. An external replication on the effects of test-driven development using a multi-site blind analysis approach. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 3:1–3:10.
- [24] Omar S. Gómez, Natalia Juristo Juzgado, and Sira Vegas. 2014. Understanding replication of experiments in software engineering: A classification. *Information & Software Technology* 56, 8 (2014), 1033–1048.
- [25] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How close are they to real faults?. In *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE Computer Society, 189–200.
- [26] Carmine Gravino, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. 2012. Do professional developers benefit from design pattern documentation? A replication in the context of source code comprehension. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science)*. Springer, 185–201.
- [27] Samir Gupta, Sana Malik, Lori L. Pollock, and K. Vijay-Shanker. 2013. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the International Conference on Program Comprehension*. IEEE Computer Society, 3–12. DOI : <http://dx.doi.org/10.1109/ICPC.2013.6613828>
- [28] M. Hammersley and P. Atkinson. 2007. *Ethnography: Principles in Practice*. Taylor & Francis.
- [29] Jo Hannay and Magne Jørgensen. 2008. The role of deliberate artificial design elements in software engineering experiments. *IEEE Transactions on Software Engineering* 34, 2 (March 2008), 242–259. DOI : <http://dx.doi.org/10.1109/TSE.2008.13>
- [30] Liang Huang and Mike Holcombe. 2009. Empirical investigation towards the effectiveness of test first programming. *Information & Software Technology* 51, 1 (2009), 182–194. DOI : <http://dx.doi.org/10.1016/j.infsof.2008.03.007>
- [31] International Organization for Standardization. 1991. *Information Technology—Software Product Evaluation: Quality Characteristics and Guidelines for their Use, ISO/IEC IS 9126*. ISO, Geneva.
- [32] ISO. 2000. *ISO 9241-11: Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs)—Part 9: Requirements for Non-Keyboard Input Devices*. ISO, Geneva, Switzerland.
- [33] ISO. 2011. *ISO/IEC 25010 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*. ISO, Geneva, Switzerland.
- [34] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. 2008. Reporting experiments in software engineering. In *Guide to Advanced Empirical Software Engineering*. Springer, London, 201–228.

- [35] N. Juristo and A. M. Moreno. 2001. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, Englewood Cliffs, NJ.
- [36] Sunwoo Kim, John A. Clark, and John A. McDermid. 1999. The rigorous generation of java mutation operators using HAZOP. In *Proceedings of the International Conference on Software & Systems Engineering and their Applications*. 9–10.
- [37] Sunwoo Kim, John A. Clark, and John A. McDermid. 2000. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of NET.OBJECTDAYS*. 9–12.
- [38] Barbara Kitchenham. 2008. The role of replications in empirical software engineering—A word of warning. *Empirical Software Engineering* 13, 2 (2008), 219–221.
- [39] B. Kitchenham, S. Pleegeer, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734.
- [40] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. DOI : <http://dx.doi.org/10.1109/TSE.2006.116>
- [41] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: A study of developer work habits. In *Proceedings of the International Conference on Software Engineering*. ACM, 492–501. DOI : <http://dx.doi.org/10.1145/1134285.1134355>
- [42] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What’s in a name? A study of identifiers. In *Proceedings of the International Conference on Program Comprehension*. IEEE CS Press, 3–12.
- [43] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering* 3, 4 (2007), 303–318.
- [44] D. T. Lykken. 1968. Statistical significance in psychological research. *Psychological Bulletin* 70 (1968), 151–159.
- [45] A. Mack and I. Rock. 1998. *Inattentional Blindness*. MIT Press. <https://books.google.com/vc/books?id=IjSjCGAG1HQC>
- [46] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2009. *An Introduction to Information Retrieval*. Cambridge University Press, England.
- [47] Manoel G. Mendonça, José Carlos Maldonado, Maria Cristina Ferreira de Oliveira, Jeffrey Carver, Sandra Camargo Pinto Ferraz Fabbri, Forrest Shull, Guilherme Horta Travassos, Erika Nina Höhn, and Victor R. Basili. 2008. A framework for software engineering experimental replications. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 203–212.
- [48] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737.
- [49] A. N. Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London.
- [50] R. Peck and J. Devore. 2011. *Statistics: The Exploration & Analysis of Data*. Cengage Learning.
- [51] S. L. Pleegeer and W. Menezes. 2000. Marketing technology to software practitioners. *IEEE Software* 17, 1 (2000), 27–33.
- [52] Filippo Ricca, Massimiliano Di Penta, Marco Torchiano, Paolo Tonella, and Mariano Ceccato. 2007. The role of experience and ability in comprehension tasks supported by UML stereotypes. In *Proceedings of the International Conference on Software Engineering*. 375–384.
- [53] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato, P. Tonella, and Corrado Aaron Visaggio. 2008. Are fit tables really talking? A series of experiments to understand whether fit tables are useful during evolution tasks. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 361–370.
- [54] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering* 30, 12 (2004), 889–903. DOI : <http://dx.doi.org/10.1109/TSE.2004.101>
- [55] Hugh Robinson, Judith Segal, and Helen Sharp. 2007. Ethnographically-informed empirical studies of software practice. *Information and Software Technology* 49, 6 (June 2007), 540–551. DOI : <http://dx.doi.org/10.1016/j.infsof.2007.02.007>
- [56] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *Proceedings of the International Conference on Software Engineering*. IEEE CS Press, 255–265.
- [57] Simone Romano, Davide Fucci, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. 2016. Results from an ethnographically-informed study in the context of test driven development. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. ACM, New York, Article 10, 10 pages.
- [58] Simone Romano, Davide Fucci, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. 2017. Findings from a multi-method study on test-driven development. *Information and Software Technology* 89 (2017), 64–77. DOI : <http://dx.doi.org/10.1016/j.infsof.2017.03.010>
- [59] Gerard Salton and Michael J. McGill. 1983. *Introduction to Modern Information Retrieval*. McGraw Hill, New York.
- [60] Felice Salviulo and Giuseppe Scanniello. 2014. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. ACM, New York, Article 48, 10 pages. DOI : <http://dx.doi.org/10.1145/2601248.2601251>

- [61] Mike Scaife and Yvonne Rogers. 1996. External cognition: How do graphical representations work? *International Journal of Human-Computer Studies* 45, 2 (1996), 185–213. DOI : <http://dx.doi.org/10.1006/ijhc.1996.0048>
- [62] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, and G. Tortora. 2014. On the impact of UML analysis models on source code comprehensibility and modifiability. *ACM Transactions on Software Engineering and Methods* 23, 2 (2014).
- [63] Giuseppe Scanniello, Carmine Gravino, Michele Risi, Genoveffa Tortora, and Gabriella Doderò. 2015. Documenting design-pattern instances: A family of experiments on source code comprehensibility. *ACM Transactions on Software Engineering and Methods* 24, 3 (2015).
- [64] Giuseppe Scanniello and Michele Risi. 2013. Dealing with faults in source code: Abbreviated vs. full-word identifier names. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 190–199. DOI : <http://dx.doi.org/10.1109/ICSM.2013.30>
- [65] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (July 1999), 557–572. DOI : <http://dx.doi.org/10.1109/32.799955>
- [66] Dan Shapiro. 1994. The limits of ethnography: Combining social sciences for CSCW. In *Proceedings of the Conference on Computer Supported Cooperative Work*. 417–428. DOI : <http://dx.doi.org/10.1145/192844.193064>
- [67] S. Shapiro and M. Wilk. 1965. An analysis of variance test for normality. *Biometrika* 52, 3–4 (1965), 591–611.
- [68] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. 2016. The role of ethnographic studies in empirical software engineering. *IEEE Transactions on Software Engineering* 42, 8 (2016), 786–804.
- [69] Helen Sharp and Hugh Robinson. 2004. An ethnographic study of XP practice. *Empirical Software Engineering* 9, 4 (2004), 353–375.
- [70] Helen Sharp, Hugh Robinson, and Mark Woodman. 2000. Software engineering: Community and culture. *IEEE Software* 17, 1 (Jan. 2000), 40–47. DOI : <http://dx.doi.org/10.1109/52.819967>
- [71] Forrest Shull, Jeffrey C. Carver, Sira Vegas, and Natalia Juristo Juzgado. 2008. The role of replications in empirical software engineering. *Empirical Software Engineering* 13, 2 (2008), 211–218.
- [72] Janet Siegmund. 2016. Program comprehension: Past, present, and future. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE (Ed.).
- [73] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering* 34, 4 (July 2008), 434–451. DOI : <http://dx.doi.org/10.1109/TSE.2008.26>
- [74] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 1997. An examination of software engineering work practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 21.
- [75] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas corpus: A curated collection of java code for empirical studies. In *Proceedings of the Asia Pacific Software Engineering Conference*. 336–345.
- [76] Sira Vegas, Cecilia Apa, and Natalia Juristo. 2016. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering* 42, 2 (2016), 120–135. DOI : <http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/TSE.2015.2467378>
- [77] M. Weiser. 1982. Programmers use slices when debugging. *Communications of the Association for Computing Machinery* 25 (July 1982), 446–452.
- [78] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer.
- [79] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. 1981. The effect of modularization and comments on program comprehension. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 215–223.
- [80] Jiaje Zhang and Donald A. Norman. 1994. Representations in distributed cognitive tasks. *Cognitive Science* 18, 1 (1994), 87–122.

Received January 2016; revised April 2017; accepted May 2017