Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance: Results from an Ethnographically-informed Study with Students and Professionals

Felice Salviulo
Dipartimento di Matematica, Informatica e Economia
Università della Basilicata
Potenza, Italy
felicesalv@hotmail.it

Giuseppe Scanniello
Dipartimento di Matematica, Informatica e Economia
Università della Basilicata
Potenza, Italy
giuseppe.scanniello@unibas.it

ABSTRACT

There are a number of empirical studies that assess the benefit deriving from the use of documentation and models in the execution of maintenance tasks. The greater part of these studies are quantitative and fail to analyze the values, beliefs, and assumptions that inform and shape source code comprehensibility and maintainability. We designed and conducted a qualitative study to understand the role of source code comments and identifiers in source code comprehensibility and maintainability. In particular, we sought to understand how novice and young professional developers perceive comments and identifier names after they have inspected the system behavior visible in its user interfaces. Novice developers were 18 third-year Bachelor students in Computer Science. The young professional developers were 12 and had work experience in between 3 months and 2 and half years. The used qualitative methodological approach is ethnographic. We asked the participants to inspect the behavior of a Java application visible in its user interfaces and then to comprehend and modify the source code of that application. We immersed ourselves and participated to the study, while collecting data by means of contemporaneous field notes, audio recordings, and copies of various artifacts. From the collected data, we have identified insights into comprehension and maintenance practices. The main insights can be summarized as follows: *(i)* with respect to novice developers, professional developers prefer to deal with identifier names rather than comments, *(ii)* all the participants indicate as essential the use of naming convention techniques for identifiers, and *(iii)* for all the participants the names of identifiers are important and should be properly chosen. Summarizing, independently from the kind of developer, it is advisable to use naming convention techniques and to properly choose identifiers.

EASE '14, May 13 - 14 2014, London, England, BC, United Kingdom Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2476-2/14/05 ... $15.00. http://dx.doi.org/10.1145/2601248.2601251

Categories and Subject Descriptors: D.2.0 Software Engineering]: General
General Terms: Experimentation, Human Factors
Keywords: Ethnographically-informed Study, Program Comprehension, Qualitative Study, Software Maintenance

1. INTRODUCTION

Many activities in software engineering involve existing software systems. Software maintenance, testing, quality assurance, reuse, and integration are only a few examples [10]. A common and crucial issue in these kinds of process entails the comprehension of source code. Several technical and managerial problems might contribute to the cost related to source code comprehensibility and maintainability [7]. For example, source code comprehensibility and maintainability may be conditioned by the developers' background and their knowledge on the application and solution domain of a subject software system as well as the available documentation produced in the analysis and design phases [1, 8, 30].

There are a number of empirical investigations on the possible benefits in terms of source code comprehensibility deriving from the use of documentation and models in the software maintenance field (e.g., [1, 15, 16, 26, 30, 32, 46]). The benefit deriving from the use of software documentation and models (if any) is quantitatively assessed by using comprehension questionnaires [16] or by the quality of the modifications performed [15]. Only few qualitative investigations have been conducted on source code comprehensibility (e.g., [29, 42, 45, 48]). For example, Roehm *et al.* [29] conducted an observational study with professional developers on the strategies they followed, the information needed, and the tools used to deal with comprehension tasks. The overall results indicate a gap between source code comprehension research and practice. In fact, the developers did not use state of the art comprehension tools and they seemed to be unaware of them. In addition, developers put themselves in the role of end-users and then explore source code.

In this paper, we present the results of a study with students and novice professional software developers to gain insight into how they deal with software maintenance and program comprehension tasks. In particular, we seek to explore

the values, beliefs, and assumptions that inform and shape source code comprehensibility and maintainability. In some sense, our work is complementary to the work by Roehm *et al.* [29] because we have investigated how developers explore source code after they have inspected the system behavior visible in its user interfaces. Given this motivation, our methodological approach is ethnographic [17].

We involved 12 professionals and 18 third-year students of the Bachelor program at the University of Basilicata in Italy. All the participants were asked to perform the following three tasks: *(i)* freely using an unfamiliar Java application to familiarize with it; *(ii)* executing some chosen usage scenarios; and *(iii)* exploring and modifying source code. While performing the task above, we immersed ourselves and participated to the tasks joining in conversations, reading the source code comments, and asking information on how the participants were facing with software comprehension and maintenance. We collected data by means of contemporaneous field notes, audio recordings of discussions, and copies of various artifacts. From the details of the data collected we sought to identify insights into the meaning of the execution of software comprehension and maintenance tasks. The results suggest the following two main results: students and professional developers found essential the use of a naming convention techniques and the choice of identifiers is important. The most remarkable difference between professionals and students was that the professionals pay more attention to the name of the identifiers than source code comments.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. The design of our study is highlighted in Section 3. In Section 4, we present our findings. The discussion of the results and possible limitations to our study are discussed in Section 5 and Section 6, respectively. Final remarks and future work conclude.

2. RELATED WORK

We first present ethnographically-informed studies in software engineering and then we discuss papers reporting on quantitative and qualitative investigations in the context of program comprehension and software maintenance.

2.1 Ethnographically-informed Studies

Ethnography has been applied in a number of ways in the context of software system design and development. For example, Beynon-Davies [4] identified a number of uses in information systems development. In particular, he 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 life cycle. Later, Beynon-Davies *et al.* [5] used ethnographic research on rapid application development to uncover the negotiated order of work in a project and the role of collective memory. Button and Sharrock [9] carried out an ethnographically-informed study in the global software development field. The goal of that study is to explain the knowledge that is displayed in the collaborative actions and interactions of design and development. Sharp and Robinson [38] reported on a study on eXtreme Programming carried out in a small company developing web-based intelligent advertisements. The main result the authors observed was that the XP developers were clearly "agile". This agility seemed intimately related to the relaxed, competent atmosphere that pervaded the developers working in groups.

Although ethnographic research has been used successfully in the software engineering field [9, 38, 47] only few studies have been conducted in the context of software maintenance [42]. Singer *et al.* [42] studied how software engineers maintain a large telecommunications system: developer's habits and tool usage during software development. This study was hosted in a single company. Despite the software engineers stating that "reading documentation" was what they did, the study found that searching and looking at source code was much more common than looking at documentation. This is the case in which there is a difference between what practitioners say they do and what they actually do. Ethnographic research might help in highlighting and explaining such a kind of discrepancies to make clearer un-remarked aspects of practice [39].

2.2 Qualitative Studies in Program Comprehension

Other researchers studied the developer behavior during the execution of maintenance tasks [14, 24, 43]. For example, LaTozza *et al.* [24] conducted a qualitative study on developers from a single company focusing on their work habits: development, maintenance, and communication. The data were gathered through surveys and interviews. 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.* [14] conducted an observational study (based on the "think aloud" protocol) on seven professional developers who updated an unfamiliar implementation of a video game. Similar to [24], the developers considered inadequate the documentation for maintenance tasks. Differently, deSouza *et al.* [13] presented the results of a study with professional developers, whose goal was to establish which kind of software artifact was the most useful when performing maintenance operations. The results suggested that source code and comments were considered the most important artifacts. The results of the three papers discussed above are very similar to that achieved by Singer *et al.* [42]: 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 and comments.

Robillard *et al.* [27] qualitatively studied five developers in a lab 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, while analyzing source code. Differently, our work sought to get some information on how developers analyze and modify source code. Therefore, our work can be considered complementary to that by Robillard *et al.* [27].

Roehm *et al.* [29] found that developers put themselves in the role of end-users, namely they perform a form of dynamic analysis. In that qualitative study, the authors also observed that program comprehension was considered a subtask of other maintenance tasks rather than a task by

itself. In addition, the results showed a gap between program comprehension research and practice: developers did not use tools for program comprehension and were unaware of them. Therefore, source code represented the only source of information to perform comprehension tasks.

Sillitto *et al.* [41] conducted a study of how developers manage change tasks in Java software using an Integrated Development Environment (IDE). The participants were nine and used the Java development environment of Eclipse[1]. The authors observed that developers explored unfamiliar source code looking for relevant entities and their relationships. There are several difference between that work and the one presented here. The main difference with respect to our study is that we performed an ethnographically-informed study into the meaning of the execution of comprehension and maintenance tasks.

Ko *et al.* [22] performed a study to understand how developers gain source code comprehension and how Eclipse might be involved. Similar to our study, the developers were unfamiliar with the source code used of the subject system. The authors found that developers interleaved three activities: searching for relevant code both manually and using search tools; following incoming and outgoing dependencies of relevant code; and collecting code and other information. The results suggested that developers need a new model of program comprehension and tools that help them to seek, relate, and collect information in a more effective and explicit manner. Differently from us, the authors did not consider the effect of comments and identifiers, while comprehending and modifying source code.

von Mayrhauser and Vans [48] studied developer's behavior during maintenance tasks. The results showed that cognition processes work at all levels of abstraction simultaneously as programmers build a mental model of the code. This study can be considered complementary to those discussed above and to that we are presenting in this paper.

Other studies examined information needed to comprehend software (e.g., [21, 40]). For example, Sillito *et al.* [40] presented two qualitative studies to understand what developers ask when performing maintenance and how they discover that information. These studies involved newcomers and professional developers. The authors identified 44 types of questions programmers ask.

Summarizing, developers considered inadequate design documentation and tools when comprehending unfamiliar source code (e.g., [24, 27]). Therefore, for the developer the only possible strategy is the analysis of the source code and/or its execution (e.g., [29]). While comprehending source code, developers ask some questions [40] to build their mental model on the system under study [48]. Accordingly, we designed an ethnographically-informed study to understand the role of comments and identifiers in source code comprehensibility and maintainability. We sought to understand how developers (novice and young) perceive comments and identifiers when comprehending and maintaining source code after they have inspected its behavior visible in its user interfaces.

[1]www.eclipse.org/

2.3 Quantitative Studies in Program Comprehension

Binkley *et al.* [6] presented a family of studies to investigate the impact of two identifier styles (i.e., *camel case* and *underscore*) on the time and accuracy of comprehending source code. The 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 the task completion time and accuracy. Later, Scanniello and Risi [34] conducted a controlled experiment on the effect of abbreviated and full-word identifier names in the comprehension of source code with the goal of identifying and fixing faults in source code. The authors observed that abbreviated identifiers provide the same information as full-word identifiers. There are several differences between the studies above [6, 34] and that reported in this paper: our study is qualitative and it identifies insights in the meaning of comprehending and modifying object oriented source code.

Takang *et al.* [44] conducted a controlled experiment based on program comprehension theories. The experiment was conducted on four versions of a program written in Modula-2 on two methods. The results suggested that commented code is more understandable than non-commented.

Studies have been also conducted to quantitatively assess the benefits of software documentation for comprehending and modifying source code (e.g., [13, 16, 46, 45]). For example, the study by Tryggeseth [46] showed the following outcomes: *(i)* the aid of having documentation available during system maintenance reduces the time needed to understand the system and modify it, and *(ii)* it also enables the maintainer with more time and better knowledge so that s/he can make more detailed changes in a restricted amount of time.

Studies to assess the effect of using the Unified Modeling Language (UML) in software maintenance have been also proposed. For example, Briand *et al.* [8] established that training is required to achieve better results when the UML is completed with the use of the Object Constraint Language. Arisholm *et al.* [1] observe that the availability of documentation based on the UML significantly improves functional correctness of changes and the design quality in case of complex tasks. Dzidek *et al.* [15] investigated the use of the UML on the costs and benefits to maintain software. The results suggest that the use of the UML significantly impacts on the functional correctness of maintenance tasks, while it does not significantly affect task completion time.

3. THE ETHNOGRAPHICALLY-INFORMED STUDY

Qualitative studies of software practice appear to be unusual in software engineering. Research in this field is dominated by quantitative studies that test hypotheses by means of laboratory experiments [36], using treatment and control groups and statistical analysis of data [49]. Differently from quantitative studies, qualitative ones allow gaining an understanding of underlying reasons and motivations behind the problem under study. Usually this kind of investigations is conducted on a small number of non-representative cases. Qualitative studies can be exploited to provide insights into the setting of a problem so generating ideas and/or hypothe-

ses for later quantitative investigations [35]. Among qualitative studies, the ethnographically-informed ones are better suited to ask questions such as *how* and *why* and *what are the characteristics of* [28]. To this end, researchers attend 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 affecting their observations [39].

In the context of our empirical investigation, we were interested in findings on: *how* novice developers and /or young professional developers perform source code comprehension and maintenance; *why* they prefer dealing with either identifiers or source code comment; and *what are the characteristics* that identifiers and source code comments must have to be help developers in the execution of comprehension and maintenance tasks.

3.1 Definition and Context

Our findings are based on a study of software comprehension and maintenance in the context of Bachelor students in Computer Science at the University of Basilicata and young professional software developers. According to the GQM (Goal Question Metrics) template [2], the goal of our study can be formalized as follows:
Analyze unfamiliar object oriented source code **for the purpose of** understanding how its identifiers and comments are used **with respect to** comprehensibility and modifiability **from the point of view of** the maintainer **in the context of** Bachelor students in Computer Science and young professional software developers.

Although the use of GQM is uncommon in qualitative studies, we decide to exploit this formalism to ensure that important aspects were defined before the planning and the execution of the study took place.

We used Bachelor because they are considered not far from novice software developers that perform small comprehension and maintenance tasks (e.g., [34]). A comparison with young professional developers would help in understanding whether and under which conditions this assumption can be considered true [11, 18].

The participants were 18 third-year students and 12 young professional software developers. The developers had work experience in between 3 months and 2 and half years. They worked in six small companies located in southern Italy. Regarding the industrial domains, one company worked in the area of software consultancy, 2 companies worked as software house/vendors, and 3 companies developed web-based systems as their main business activity.

All the participants (professionals and students) were volunteers. The students accomplished the study as a part of a series of optional laboratory exercises conducted within a Software Engineering course. During the Bachelor program, the participants had passed all the exams related to the following courses: Procedural Programming, Object Oriented Programming I and II, and Web Development. In these courses, the students performed assignments about the development of object-oriented software systems and web-based applications. In the object oriented programming courses, the students used NetBeans[2] as the IDE.

The professional developers were employees of software companies in the industrial contact network of our research group. The grater part of them took the Bachelor or the Master degree in Computer Science from the University of Basilicata. The professionals participated in the study outside of their work hours. This choice allowed us to have motivated participants. These participants had previously used Eclipse, NetBeans, and Visual Studio[3].

All the participants were aware of the pedagogical and practical purposes of the study, while they did not know its goal. The study results did not influence the grades the students got for their Software Engineering course. As for professional, we did not pay them for their participation.

3.2 Experimental Object

We used *guess my number* as the experimental object. It is a game whose object is to use clues to figure out a secret number. In particular, that game randomly chooses a secret number and the user is asked to guess this number. If the guess is too high or too low, the application gives the user a hint (i.e., your guess is too high or too low). The game was implemented in Java and was based on the MVC (Model-View-Controller) architectural model. The total number of classes in guess my number was 20, while the source lines of code (SLOC) were 853, and the comment lines of code (CLOC) were 543.

Constant variables in source code were written in uppercase characters separated by underscores (`CONSTANT_VARIABLE`). The names of methods and classes were specified using the naming conventions lower (`camelCase`) and upper camel case (`CamelCase`), respectively. Variables were specified in lower camel case. In addition, the names of identifiers (class and method names and variables) were short yet meaningful and mnemonic. That is, each name clearly indicates the intent of the identifier usage. One-character variable names were only used for temporary variables. The compounding words of the identifiers were in Italian language and were not in abbreviated form.

Source code comments were present for each class and for each method. Comments were in Italian language and were mostly used to summarize code and to explain the programmer's intent. The comment formatting was consistent with the Javadoc standard. Compounding words of identifiers and the source code comments were in Italian because different participants may have different familiarity with English. This concern could be difficult to control and could potentially bias results.

The software application was chosen to fit the time constraints (i.e., participants gave their availability for a fixed amount of time, namely two hours) of the study though realistic for small/medium comprehension tasks and small corrective maintenance operations [31]. It is worth mentioning that for the professional developers we did not use the source

[2] netbeans.org
[3] www.microsoft.com/visualstudio/

code they developed for two main reasons: *(i)* we were interested in studying the effect of identifiers and comments of unfamiliar source code on comprehensibility and modifiability and *(ii)* software companies are reluctant to furnish their own source code also for research purposes. The use of commercial applications is however the subject of our future work. In fact, we plan to involve some companies of our industrial contact network in a research project on the topics concerned with the study presented here. This point is the most challenging phase of our research plan.

The experimental package and the gathered data can be downloaded at www2.unibas.it/gscanniello/SCCMEthno/.

3.3 The Study

The study was conducted by a single observer (one of the authors of this paper) in between January and February 2013 and was founded on one-to-one sessions between the observer and the participants. This experimental procedure is customary in ethnographically-informed studies (e.g., [28]). We conducted our study in Italian to avoid biasing the study results. Different participants 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 guess my number. 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 [39]; *(ii)* the observer could appreciate the perspective of the developers while carrying out the tasks assigned; and *(iii)* the observer could get information on how the IDE (i.e., NetBeans) was used. While conducting the study the observer avoided influencing the participants during the interaction.

The data were collected using: contemporaneous field notes, audio recordings of discussions, and copies of various artifacts (e.g., source code). The audio recordings of discussions and the gathered data are available on the web page where we made available the experimental package.

3.4 The Setting

We describe the physical setting of the study because it is a good practice in ethnographically-informed studies and because the spatial organization could be relevant insofar as the developers work to accomplish the tasks [38]. In particular, professional developers conducted the study in their own office with their own personal computer (or laptop), while the students conducted it in a didactic laboratory at the University of Basilicata with their own laptop. The office of all the professional developers was open-plan, having an overall rectangular shape. The number of desks for each office was in between 4 and 8. The design choices above were taken to reduce as much as possible biases concerned to the physical setting of the study (e.g., scant familiarity with the personal computer or with the place were the experimental session took place).

3.5 Design

The study was organized in the steps below:

1. *Illustrating the study.* The observer introduced the study to each participant. For each participant, the observer tried to reproduce the same introduction to the study using a prearranged schema. The observer did not provide details on the research hypotheses of the study.

2. *Filling in a pre-questionnaire.* Each participant was asked to fill in the pre-questionnaire to gather some information about their industrial working experience, grade point average, and knowledge on NetBeans. We opted for NetBeans because we expected that the participants had a high level of familiarity with this IDE. Although this choice could affect the results, we believe that the use of a different IDE or allowing the participants to choose their preferred IDE could even more threaten the validity of the validity of the results.

3. *Inspecting the application behavior visible in the user interfaces.* In this step, the participants accomplished two tasks. In the former, they freely used the Java application. In the second task, the observer asked the participants to execute the application according to an usage scenario (see Table 1) that was the same for all the participants in the study. The observer took note of the time needed to accomplish these tasks and of the time to complete the steps 4 and 5. To reduce possible apprehension concerns, the participants were not aware that the observer has been taking note of the time to accomplish the two tasks and the time to execute the steps 4 and 5.

4. *Comprehending source code.* The observer first asked the participants to freely browse the source code to become familiar with it and to increase the knowledge of the solution domain of the application understudy. Later, the participants were asked to identify and analyze the source code concerning: *(i)* the generation of the secret number; *(ii)* the input of the guesses; and *(iii)* the visualization of winnings records.

5. *Modifying source code.* In the used implementation of guess my number, the number of attempts was not limited. Therefore, we formulated and asked to each participant to implement the following change request: modify the source code of guess my number such that the maximum number of attempts to guess the secret number is five. The participant could move to the next step when the change request was implemented. It is worth mentioning that the role of the observer was relevant in this step. In fact, he engaged with each participant, but without conditioning his/her work habit.

6. *Filling in a post-questionnaire.* It aimed at gaining some insights to strengthen and explain the results.

Table 1: Usage scenario of guess my number

Stage	Description
1	Insert your name and start the game
2	Insert two attempts to guess the secret number
3	Interrupt the game
4	Show the winnings records

Table 2: Post-experiment Survey Questionnaire

Id	Question
Q1	What did you like of guess my number?
Q2	What did you dislike of guess my number?
Q3	What did you find unclear and/or difficult while using guess my number?
Q4	Did you find source code comments useful to execute the tasks?
Q5	What did you dislike of the source code comments?
Q6	What did you find unclear or difficult in the source code comments?
Q7	Do you have some ideas on how to modify source code comments to improve source code comprehensibility and modifiability?
Q8	What did you dislike of the source code?
Q9	What did you find unclear or difficult in the source code?
Q10	Do you have some ideas on how to modify source code to improve its comprehensibility and modifiability?
Q11	Do you believe that the use of lower and upper camel case naming conventions for the identifiers improves source code comprehensibility and modifiability?
Q12	What do you think of NetBeans?
Q13	Please compare NetBeans with the other IDEs you know and use.

The post-experiment survey questionnaire is shown in Table 2. All the questions expected open answers.

In classic ethnography, researchers study practice by immersing themselves in the context under study for a long time. In the software engineering field, this is not always possible because of time constraints and/or of commercial confidentiality. Therefore, it has become quite common to adapt ethnographic traditions to conduct shorter studies [28]. In the ethnographically-informed study presented here, the observer immersed himself and participated in the steps 3, 4, and 5 – joining in conversations, reading the artifacts, etc. He did not disturb or change the natural setting. An informal approach was used to probe possible issues in a naturalistic manner [25]. It was not easy to immerse in the experimental context with the objective of generating scientific knowledge and providing relevant results. This was one of the most important issues to hold.

The observer also provided support for the steps 1, 2, and 6. If necessary, he clarified concerns related to the study and/or to the questions of both the pre- and post-questionnaires.

4. FINDINGS

Our analysis followed a standard ethnographic approach (e.g., [5]). In particular, we turned around the identification of the main themes emerged from our data and gathered artifacts.

4.1 Ethnographic Analysis

The goal of an ethnographic analysis is to find insights in the form of recurrent themes. In other words, such a kind of analysis tries to identify insights into the meaning of certain activities from the details of the data collected [38]. To find the insights of our ethnographically-informed study, the observer reflected on the experience gained in the immersion and used all of the data to recollect, revisit, and reconsider what was found. This took place alone first and then through a discussion with the other researcher. The discussion was based on the audio recordings, the answers to the post-experiment survey questionnaire (shown in Table 2) and the copies of the various artifacts (e.g., the source code modified by the participants).

When a theme appeared to be emerging in a group (i.e., students or professional developers), we searched data in the same group that could contradict this theme. If no contradictory evidence emerged then the theme was pursued. The analysis proceeded iteratively: themes were identified, dropped or validated and then confirmed. This approach required a considerable effort especially in the validation of the potential themes with respect to the data collected. Potential themes were identified while conducting the study and after its conclusion.

In the following subsections, we illustrate and detail the themes confirmed in our ethnographic analysis.

4.1.1 Application Behavior

When the participants used the application, they carefully inspected its user interfaces. They appreciated the fact that guess my number provided a number of aids to facilitate its use: keyboard and mouse accelerators. We noted that these tools were appreciated by all the participants in the study and that they were passionate about using and interacting with the application. In other words, the way the participants approached comprehension and maintenance tasks was positively affected by the usability of the application under study. We also observed that there was a desire to produce usable software. The findings were the same independently from the group of participants (i.e., students and professional software developers).

4.1.2 Comments

We observed a difference between professional software developers and students. In particular, to comprehend and to maintain source code professionals either did not read comments or quickly scanned them. Differently, students first carefully read the comments of the class to which he/she was interested in and then those of its methods. In this context, it is easy to imagine that comments are vital for less experienced participants, while they are not for more experienced ones. A possible interpretation for this result could concern the fact that professional developers have learned to not trust source code comments because they are very often not updated during the software maintenance/evolution and therefore may be misleading. This point was clear when the participants modified the application to deal with the change request (see the step 5 in Section 3.5). In particular, students carefully modified the existing comments and added the comments to the newly written source code. That is, producing quality comments seemed to be of primary importance for students. Differently, professionals neither updated nor added comments to the source code of the application under study when modifying it as a consequence of the implementation of the given change request.

4.1.3 Source Code

Producing quality code was of primary importance for students and professional developers and was alluded to in different ways. Although we did not explain the standard and

the naming convention to be used for modifying the source code of guess my number, each developer/student was able to pick up the code they were working on extremely quickly. In this concern, we did not observe any remarkable difference between professional developers and students even if it seemed that professional developers quicker became comfortable with the application and its source code. This finding was partially confirmed by the qualitative analysis we reported in Section 4.2.

Similar to Ko *et al.* in [22], we noted that the participants first looked at how the classes were organized in packages and sub-packages and then they inspected the source code following incoming and outgoing dependencies of relevant code. We also observed that the text of both the keyboard shortcuts and the tooltips messages were used by the participants (especially professional developers) to search for relevant code. This result suggests that developing for usable applications might also positively influence how maintainers browse and comprehend source code before executing a maintenance operation.

All the participants used their own coding style (e.g., indentation) for the code added as a consequence to the given change request. For that code, the participants easily used the camel case naming convention. This might indicate that: camel case is considered as the standard for novice and young Java programmers or developers are flexible enough to adopt the convention that was used in the code. This interesting point deserves future and special conceived empirical investigations.

4.1.4 Using NetBeans

With respect to students, the professional developers better used the functionalities of the used integrated development environment (i.e., NetBeans) to browse source code and debug it. We observed that the familiarity with NetBeans made the difference especially when the participants had to modify the source code to deal with the change request. This result was corroborated by the quantitative analyses we summarize in the next subsection.

4.2 A Quantitative Look Inside the Study

We quantitatively analyze the differences between professional developers and students with respect to the time they spent to accomplish the steps 3, 4, and 5 of our study (see Section 3.5). To this end, we used statistical analyses. In particular, the tested null hypotheses, described here as a single parameterized hypothesis, are:

Hn_Step_i The time (expressed in minutes) the professional developers spent to accomplish the step i (i.e., 3, 4, and 5) is not significantly less than that of the students.

The hypotheses are all one-sided because we expected a positive effect of experience on the time (i.e., the dependent variable) to comprehend and/or modify source code. In other words, more experienced participants should spend less time to accomplish each step. We did not perform this further analysis to contribute to the answer of our research question rather then to give strength (if needed) to the achieved qualitative results.

To test the null hypotheses, we used unpaired statistical tests. In case of normally distributed data, we opted for an unpaired t-test. We verified the normality of the data by means of the Shapiro-Wilk W test [37]. If the data were not normally distributed, we opted for the Mann-Whitney test (also known as Wilcoxon rank-sum test) [12]. While the statistical tests allow for checking the presence of significant differences, they do not provide any information about the magnitude of such a difference. To this end, we computed effect size. In particular, we used the Cohen's d and the Cliff's d [19] in case of parametric and non parametric analyses, respectively.

The results of the the Mann-Whitney test (the data were not normally distributed) did not allow to reject the null hypotheses Hn_Step_3 and Hn_Step_4. The effect size is small in both the cases. The values of the Cliff's d are -0.185 and 0.176, respectively. That is, the effect size can be considered small [19] and no remarkable differences between professionals and students were observed in these steps.

The results of an unpaired t-test allowed rejecting Hn_Step_5 (p-value < 0.001). The effect size is large (Cohen's $d = -1.29$). This means that more experienced participants spent significantly less time to modify the source code of guess my number.

The findings show a clear tendency: more experienced participants (professional developers) spend less time than less experienced ones (students) to execute comprehension and modification tasks on unfamiliar source code. This difference is statistically significant when source code has to be modified as a consequence of a change request. Even if these findings can be considered somewhat expected, it is acceptable as commonplaces need to be verified through empirical studies [3, 20].

5. DISCUSSION

The "So What?" factor is relevant in empirical software engineering and ethnography, in particular. That is, what significance do the results have for software development? [38]. One of the main goals of ethnographically-informed study is to uncover implicit features of practice. In the context of the study presented here, what do the achieved results tell us about maintenance, in general? And what do the achieved results tell us about comprehensibility and modifiability of source code? First, our results show that professional developers do not look at the comments, but prefer to deal with source code statements while performing comprehension and maintenance tasks. To comprehend unfamiliar source code, professional developers are less interested in the comments than the source code itself. This result also holds when they have to write source code as a consequence of a change request: they carefully chose the identifiers and neither wrote nor updated comments. The results above might be important from both the professional and the researcher perspectives. For the professional, the message is choose good names for the identifiers. The researcher could be interested in investigating the professional developer's mental model to comprehend and modify source code, when dealing with well written source code and comments. In addition, the researcher could find useful the findings above because these could affect the previous research in software maintenance

and evolution (e.g., software clustering, concept location). A number of approaches in these fields assume that source code is adequately commented (e.g., [23, 33]). This could represent an issue if professional developers do not properly comment source code. Then, our findings call for reconsidering research agendas in the context of lexical based approaches for program comprehension and software maintenance.

As for the students, they gave the same importance to comments and source code statements. A plausible explanation for this finding is that student are less experienced with programming and less accustomed to comprehend and modify source code written by other students/developers. From the researcher perspective, this result and those above are interesting because it would be useful to understand what should be the experience level a developer need to effectively comprehend and maintain source code that is badly commented.

Both professionals and students found the use of naming convention techniques for software identifiers as essential to effectively perform comprehension and maintenance tasks. In addition, the findings also suggested that both the kinds of developers preferred identifiers whose names are short, meaningful, and mnemonic. This result is relevant from the professional perspective. In fact, it is better to spend some effort to properly choose the names of identifiers and to write readable source code than to spend some effort to update and to write clear and meaningful comments. This could not be valid for less experience developers. However, due to the low difference in the experience of the two kinds of participants in our study, we can suggest that it is better to focus on source code statements rather than on source code comments, while developing or maintaing object oriented source code.

The results also show that professional software developers are more familiar with the features of NetBeans. In particular, the use of the tools for searching and looking at source code is more fluid for professional developers. Similar results were obtained for the use of the debugging tool. These findings are clearly related to the larger experience and familiarity the professional developers had with IDEs. Before the study took place, they had used the following IDEs: NetBeans, Eclipse, and Visual Studio. On the other hand, all the students knew NetBeans and only a few of them Eclipse and Visual Studio.

The obtained results pose the base for future investigations (both quantitative and qualitative) on how developers comprehend and maintain source code. For example, it could be meaningful to investigate whether the presence of source code comments improves or not source code comprehensibility and maintainability. Another possible future direction four our research could concern the effect of using abbreviated and expanded identifier names on specific tasks related to program comprehension. The results obtained in the study presented here and those presented by Scanniello and Risi [34] pose the basis for such a kind of studies.

6. LIMITATIONS

In this section, we will discuss possible limitations for our empirical investigation.

6.1 External Validity

The *external validity* threats could be present in case an empirical study involves students as the participants [11]. However, the students can be considered representative of the context where we would like to generalize the results (i.e., novice software developers that perform small comprehension and maintenance tasks). Working with students also implies various advantages, such as the students' prior knowledge is rather homogeneous. Nonetheless, in order to strengthen the external validity, we also involved professional developers. Another threat to external validity could concern the size and the domain of the used application. This could have affected the outcome that professionals mostly ignored comments. In fact, comments might become more important for professional developers in case of larger and more complex applications. This point deserves special conceived future empirical investigations. The use of NetBeans represents another possible threat to the validity of the results. The use of a different IDE could lead to different results. The use of experiment material in Italian language could also lead to external validity threats.

6.2 Construct Validity

It is related to some social factors (e.g., evaluation apprehension). To mitigate this threat the observer immersed himself in the execution of the study and used an informal approach to interact with the participants. In addition, the students were not evaluated on the results they achieved in the study. Other threats could be related to the programming experience of the participants, the used Java application, and the post-experiment questionnaire used. Finally, the use of a single observer and the influence that he could practice on the participants may also affect the validity of the results.

6.3 Conclusion Validity

It concerns the selection of the participants. We drew fair samples and conducted our study with participants of these samples. Conclusion validity could be also affected by the number of participants in the study. To mitigate this threat, we plan to conduct replications on a larger number of participants. Since we also performed quantitative analyses, statistical tests represent another possible threat to conclusion validity. We used well known and widely used parametric and non-parametric tests. It was also explicitly mentioned, when no significant difference was present.

6.4 Internal Validity

This kind of threat has been mitigated since the students in each group had similar experience with Java and NetBeans. This was not exactly the same for professional developers.

7. CONCLUSION

In this paper, we report on an ethnographically-informed study conducted to investigate how students and young professional developers perform source code comprehension and maintenance tasks on object oriented source code. Our study was ethnographically-informed in two particulars:

- The essential nature of practices (comprehension and maintenance of source code in absence of software documentation) is not known. It needs to be verified through appropriate empirically studies.

- Our focus is on the practices of software comprehension and maintenance, as it is carried out in the real world in natural settings without perturbing them. We did not influence the execution of the tasks and collected our data through: observations of practice, discussions with participants, and analysis of the produced artifacts (e.g., the modified source code). We avoid any form of control and intrusion, such that all our data were as much as possible naturally occurring.

From the collected data, we have identified and confirmed some themes that can be summarized in the following results: *(i)* professional developers use better the features of NetBeans to browse source code and debug it; *(ii)* professional developers pay more attention to the name of identifiers; *(iii)* students and professional developers indicate as essential the use of a naming convention techniques for source code identifiers; and *(iv)* the choice of the identifier names is important. These achieved results pose the bases for a number of future investigations in the context of source code comprehensibility. For example, it could be meaningful to quantitatively investigate whether and how the presence of source code comments improves or not source code comprehensibility and maintainability.

To further verify the achieved results, we plan to replicate our study with different participants from different contexts. The use of different applications represents another future direction four our research. Future work will be also devoted to study whether the achieved results also hold in the context of programming languages different from Java. An further interesting future direction for our research could be to evaluate how code and comments co-evolve, and if that could be solved in a semi-automated way.

Acknowledgment

We thank the students and the professional developers for their participation in our ethnographically-informed study.

8. REFERENCES

[1] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Trans. on Soft. Eng.*, 32:365–381, 2006.

[2] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[3] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng.*, 25(4):456–473, 1999.

[4] P. Beynon-Davies. Ethnography and information systems development: Ethnography of, for and within is development. *Information & Software Technology*, 39(8):531–540, 1997.

[5] P. Beynon-Davies, D. Tudhope, and H. Mackay. Information systems prototyping in practice. *Journal of Information Technology*, 14(1):107–120, Mar. 1999.

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

[7] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[8] L. C. Briand, Y. Labiche, M. Di Penta, and H. Yan-Bondoc. An experimental investigation of formality in UML-based development. *IEEE Trans. on Soft. Eng.*, 31(10):833–849, 2005.

[9] G. Button and W. Sharrock. Project work: The organisation of collaborative design and development in software engineering. *Computer Supported Cooperative Work*, 5(4):369–386, 1996.

[10] E. Byrne. Software reverse engineering. *Software, Practice and Experience*, 21(12):1349–1364, 1991.

[11] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *Proceedings of the International Symposium on Software Metrics*, pages 239–, Washington, DC, USA, 2003. IEEE Computer Society.

[12] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[13] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the International Conference on Design of communication: documenting & designing for pervasive information*, pages 68–75. ACM, 2005.

[14] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 183–192. ACM, 2005.

[15] W. J. Dzidek, E. Arisholm, and L. C. Briand. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Trans. on Soft. Eng.*, 34:407–432, May 2008.

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

[17] M. Hammersley and P. Atkinson. *Ethnography: Principles in Practice*. Taylor & Francis, 2007.

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

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

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

[21] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

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

[23] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3):230–243, 2007.

[24] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 492–501. ACM, 2006.

[25] C. Passos, D. S. Cruzes, T. Dybå, and M. Mendonça. Challenges of applying ethnography to study software practices. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '12, pages 9–18. ACM, 2012.

[26] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, June 2002.

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

[28] H. Robinson, J. Segal, and H. Sharp. Ethnographically-informed empirical studies of software practice. *Inf. Softw. Technol.*, 49(6):540–551, June 2007.

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

[30] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, and G. Tortora. On the impact of UML analysis models on source code comprehensibility and modifiability. *ACM Trans. on Soft. Eng. and Meth.*, (to appear), 2013.

[31] G. Scanniello, C. Gravino, and G. Tortora. Investigating the role of UML in the software modeling and maintenance - a preliminary industrial survey. In *Proceedings of the 12th International Conference on Enterprise Information Systems*, pages 141–148, 2010.

[32] G. Scanniello, C. Gravino, and G. Tortora. Does the combined use of class and sequence diagrams improve the source code comprehension? results from a controlled experiment. In *Proceedings of the International Workshop Experiences and Empirical Studies in Software Modelling*, pages 4:1–4:6. ACM Press, 2012.

[33] G. Scanniello and A. Marcus. Clustering support for static concept location in source code. In *Proceedings of International Conference on Program Comprehension*, pages 1–10, 2011.

[34] G. Scanniello and M. Risi. Dealing with faults in source code: Abbreviated vs. full-word identifier names. In *Proceedings of International Conference of Software Maintenance*. IEEE Computer Society, 2013.

[35] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, July 1999.

[36] J. Segal, A. Grinyer, and H. Sharp. The type of evidence produced by empirical software engineers. In *Proceedings of the workshop on Realising evidence-based software engineering*, pages 1–4. ACM, 2005.

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

[38] H. Sharp and H. Robinson. An ethnographic study of xp practice. *Empirical Softw. Eng.*, 9(4):353–375, 2004.

[39] H. Sharp, H. Robinson, and M. Woodman. Software engineering: Community and culture. *IEEE Softw.*, 17(1):40–47, Jan. 2000.

[40] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, July 2008.

[41] J. Sillito, K. D. Volder, B. Fisher, and G. C. Murphy. Managing software change tasks: an exploratory study. In *Proceedings of International Symposium on Empirical Software Engineering*, pages 23–32. IEEE Computer Society Press, 2005.

[42] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, pages 21–. IBM Press, 1997.

[43] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, Sept. 1984.

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

[45] S. Tilley and S. Huang. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *Proceedings of the annual international conference on Documentation*, pages 184–191. ACM Press, 2003.

[46] E. Tryggeseth. Report from an experiment: Impact of documentation on maintenance. *Empirical Software Engineering*, 2(2):201–207, 1997.

[47] S. Viller and I. Sommerville. Ethnographically informed analysis for software engineers. *Int. J. Hum.-Comput. Stud.*, 53(1):169–196, 2000.

[48] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Trans. Softw. Eng.*, 22(6):424–437, June 1996.

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