Intl. Journal of Human–Computer Interaction, 29: 743–763, 2013

ISSN: 1044-7318 print / 1532-7590 online
DOI: 10.1080/10447318.2013.773876

Taylor & Francis
Taylor & Francis Group

How Does Software Visualization Contribute to Software Comprehension? A Grounded Theory Approach

Hacı Ali Duru[1], Murat Perit Çakır[2], and Veysi İşler[2]
[1]*Turkish Military Academy, Ankara, Turkey*
[2]*Middle East Technical University, Ankara, Turkey*

Despite their ability to synthesize vast amounts of information, software visualization tools are not widely adopted in the software engineering industry. In an effort to investigate the underlying reasons, we conducted a usability study to investigate the affordances of software visualization techniques for the maintenance of complex software systems. Expert programmers were asked to carry out programming tasks with or without using a software visualization tool while their screens and eye gaze patterns were recorded. Statistical analysis of task performance data showed that participants who used the software visualization tool outperformed the control group in terms of task completion time and accuracy. However, quantitative analysis of performance measures did not reveal in what ways software visualizations contributed to this improvement. In an effort to identify the cognitive strategies that underlie this quantitative performance difference, process models grounded in qualitative analysis of eye-tracking data were constructed. The process models indicated that software visualizations guided the subjects in the experiment group toward following specific software comprehension strategies, which account for the difference observed in task performance data.

1. INTRODUCTION

Modern software systems have considerably grown in size, become highly integrated across various platforms, and hence exhibit an increasingly complex structure. Given the fact that software industry has a high turnover rate and software is often obliged to provide uninterrupted service in a competitive environment, maintaining software has become increasingly difficult over the past years. In particular, up to 80% of the software life cycle costs are maintenance related, and nearly half of that maintenance budget is used for *understanding the source code* (Telea, Ersoy, & Voinea, 2010).

We thank the TAF-METU Modeling and Simulation Center and the METU CEIT Eye Tracking Lab for their support and to SMACCHIA.COM S.A.R.L for making the full version of NDEPEND available to us to conduct a usability study.

Address correspondence to Murat Perit Çakır, Department of Cognitive Science, Informatics Institute, Middle East Technical University, Universiteler Mah., Dumlupınar Bulvarı No:1, Cankaya, Ankara, 06800, Turkey. E-mail: perit@metu.edu.tr

Jun, Landry, and Salvendy (2011) stated that "the goal of computer supported and interactive visual representations is to amplify cognition where they can help facilitate analytical reasoning, support decision making, and allow users to gain insight into complex problems" (p. 348). As a branch of Visual Analytics, software visualization tools also allow users to synthesize and make sense of vast amounts of information regarding the inner organization of software modules and their interaction with each other. Such information is vital for conducting high-cost operations such as software maintenance, which requires a thorough understanding of existing source code. However, despite the high percentage of maintenance costs, the utility of software visualization in the industry is still in question. In particular, whether software visualization has a positive effect on software comprehension has been a controversial topic (Bessey et al., 2010; Telea et al., 2010; Umphressa, Hendrixa, Cross, & Maghsoodloob, 2006).

Revealing how software visualization may contribute to software understanding requires a systematic investigation of the relationship between cognitive processes underlying software comprehension and the software visualization techniques. Usability studies that focus on this relationship tend to rely on task performance metrics and retrospective questions to assess the level of comprehension aided by visualizations. Kagdi, Yusuf, and Maletic (2007) termed such measures as "black box" evaluations because they capture only the final outcome of the software comprehension process. These measures are useful for identifying whether software visualization has a positive effect on software comprehension. However, such black box evaluations do not reveal how visualizations are used in practice and how their use may lead to a deeper understanding of a source code's structure and organization.

Software visualizations aim to display structures within the source code that may not be self-evident, so that programmers can notice and attend to them. Thus, it is important to observe how particular visualization strategies direct programmers' attention in a complex information space. Eye-tracking studies that focus on the process in which visualizations are used have the potential to reveal the details of the software comprehension process mediated by visualizations. However,

despite this potential, there are only a few studies that utilize eye-tracking technology in this context (Bednarik & Tukiainen, 2006). Existing eye-tracking studies tend to rely on quantitative measures derived from eye-tracking data, such as fixation sequences and scan paths elicited by the visualizations while completing a programming task. For instance, Kagdi et al.'s (2007) study focuses on the fixations and scan-paths elicited by Unified Modeling Language diagrams while programmers were working on them. Such quantitative features are useful in terms of revealing microlevel details of visual information processing and which areas of interest received the most attention. However, scaling this microlevel analysis up to the comprehension of broader programming structures that are particularly relevant to more realistic software maintenance task scenarios remains to be a challenge.

In this study we used eye-tracking technology to record expert programmers' eye gaze patterns and screen actions while they completed seven software maintenance tasks with or without using a software visualization tool. The software to be maintained was a web-based e-commerce application that is implemented in the Microsoft .Net framework. In an effort to address the methodological gap between outcome measures and the microlevel features provided by the eye tracker, we employed a grounded theory approach to conduct a qualitative analysis of the various ways our subjects attempted the tasks in both conditions. Based on our qualitative analysis of the screen recordings overlaid with fixation sequences, we constructed process models of each group's activities in an effort to identify the differences between the two interface conditions. Statistical analysis of outcome measures showed that the experimental group that made use of the software visualization tool outperformed the control group. Most important, our qualitative analysis suggests that this difference is due to the affordances of the visualizations, which guided the subjects in the experiment group toward following effective software comprehension strategies that ultimately contributed to their success.

The rest of this article is structured as follows. Section 2 provides a background on related work in the software comprehension and visualization literature. Section 3 introduces the materials and the methods used in the study. Section 4 presents the findings of our quantitative and qualitative analyses. Finally, the article concludes with a discussion of these findings.

1.1. Related Work

Software maintenance requires programmers to comprehend parts of the source code relevant to their task. Because this study is concerned with identifying the contribution of software visualization techniques to software maintenance processes, a review of related literature in software comprehension and software visualization is provided below to set the conceptual background of the study.

2. SOFTWARE COMPREHENSION

Corritore and Wiedenbeck (1991) defined software comprehension as "identifying important program parts and inferring relationships between them" (p. 199). Software comprehension literature primarily focuses on identifying the types of cognitive strategies programmers employ while they try to solve programming-related problems. Studies conducted with software maintainers have found that programmers use different program comprehension strategies depending on a number of parameters, such as level of expertise, programming style, personality characteristics, the nature of the program, and the programming language used (Karahasanovic, Levine, & Thomas, 2007).

Most frequently mentioned program comprehension strategies in the literature involve top-down, bottom-up, and interactive approaches. In the top-down strategy, programmers devise assumptions about the overall objectives of the program and then investigate every subsection of the program in light of those holistic assumptions (Brooks, 1983; Soloway, Adelson, & Ehrlich, 1988). In the bottom-up model, programmers start by reading code statements until they construct a high-level mental representation of the program (Karahasanovic et al., 2007). Finally, interactive models combine aspects of top-down and bottom-up strategies during software comprehension. For instance, von Mayrhauser and Vans (1996) argued that understanding of software is developed simultaneously at several levels of abstractions while programmers freely switch between top-down, bottom-up, and knowledge-based strategies (Storey, Wong, & Muller, 2000).

Novelty of the software is another important factor that influences the software comprehension strategies chosen by the programmers. For instance, Shaft and Vessey (1995) and von Mayrhauser and Vans (1996) proposed that programmers employ a top-down, goal-oriented, hypothesis-driven approach for software comprehension while they are working in a familiar domain in which they recognize a large number of plans or design patterns. On the other hand, Corritore and Wiedenbeck (2001) found that programmers tend to use the bottom-up strategy if the program is novel to them. These authors also report that in an industrial-sized software consisting of tens or hundreds of thousands of lines of code, programmers may switch between top-down and bottom-up approaches frequently within the same program, because their state of knowledge about the domain varies across different parts of the software (Corritore & Wiedenbeck, 2001).

The nature of the maintenance task carried out on a program and the cognitive complexity of the software also influence which comprehension strategies are employed by the programmers (Koenemann & Robertson, 1991). For instance, programmers tend to use the bottom-up strategy in a task like bug fixing, whereas when they perform a task such as adapting software to a new platform, programmers tend to use the top-down approach (von Mayrhauser & Vans, 1996).

The interplay between the type of the task and the complexity of the software requires programmers to develop a level of understanding that will be adequate for the purposes at hand. This led Littman, Pinto, Letovski, and Soloway (1986) to propose two comprehension strategies with respect to the breadth of comprehension they entail, namely, the "systematic" and the "as needed" approaches. In the systematic approach programmers try to develop a broad understanding of the program. The authors claim that programmers who use this approach tend to be more successful in accomplishing most maintenance tasks. In the as needed approach, programmers don't aim to understand the whole design and pragmatically attend to only those parts that are deemed relevant to the task at hand.

Our review suggests that successful software maintenance is closely related to effective use of software comprehension strategies. It is essential for a programmer to develop an adequate understanding of how the software carries out its main functionalities as well as how the modular organization of the code and the dependencies built in between components realize those functionalities. Software comprehension literature primarily focuses on how programmers develop such an understanding by using conventional tools offered by typical software development environments. Our review identified some of the strategies frequently used by programmers in response to various factors such as the cognitive complexity and the size of the source code as well as the specifics of the task and the programming environment used. The strategies covered in this section are used as an explanatory framework to characterize the participants' actions we observed during our experiment. In the next section, we review software visualization techniques that aim to help programmers execute such strategies in an effective way.

2.1. Software Visualization

Software visualization is a special application area of visual analytics that is concerned with the interactive analysis of software mediated by graphical tools and representations (Grant, 1999). Software visualization techniques may involve the use of interactive computer graphics, animation, cinematography, and visual arts to aid the understanding of computer programs (Price, Baecker, & Small, 1993). Comprehensive reviews of software visualization literature identified program and algorithm visualization as the two main threads among existing software visualization applications (Lemieux & Salois, 2006; Price et al., 1993).

Program visualization refers to the visualization of data structures and their dependencies with static and/or dynamic representations. Static code visualizations provide a graphical demonstration of the routines and the control flow in software (Wettel & Lanza, 2008). Such visualizations usually rely on certain metaphors to aid the programmers' comprehension of the source code. For example, Wettel and Lanza (2008) used the city metaphor by symbolizing the program units as city blocks and buildings. Dynamic visualizations animate such static representations by dynamically coloring or altering them. For instance, the source code is colored or highlighted while the program is running to highlight what part of the code is executed at a given time and what values are bound to specific variables of interest. Such animated visualizations aim to help programmers observe how the information flows among the modules that constitute the source code. Finally, algorithm visualization is concerned with providing a high-level description of what a program does, rather than the specific way a program is implemented in a specific language (Lemieux & Salois, 2006). In other words, algorithm visualization aims to reveal the reasoning realized by the sequence of actions that define the functionality of the software at an abstract level. Therefore, the visualizations used may not necessarily correspond to actual data or instructions implemented in software.

Several studies have investigated in what ways software visualization tools influence software comprehension processes, albeit with conflicting results (Storey et al., 2000, Voinea, Telea, & Wijk, 2005). For instance, the study conducted by Petre (2010) considers the relationship between mental imagery and software visualization in a professional, high-performance software development context. Petre observed that experts tend not to use generic visualization tools, because they do not sufficiently reveal the design rationale underlying the organization of source code. Petre also reported that experts use custom visualization tools that they think better embody their domain knowledge, which may differ from the tools other experts use. Telea et al.'s (2010) study also suggests that software visualization tools have limited adoption in the IT industry. The study relied on interview data gathered while authors participated in several software development projects. Stakeholders' comments suggested that they had different software visualization needs which were not adequately supported by existing visualization tools.

There are also empirical findings that highlight the positive contribution of software visualization on software comprehension. Koschke and Diehl's (2002) study provides strong evidence in favor of the utility of software visualization tools in the context of reverse engineering. Moreover, Umphressa et al. (2006) reported that control structure diagrams and the complexity profile graph that abstract away the intricacies of the source code positively affect program comprehension. Such studies highlight the potential of software visualization techniques in task contexts where programmers need to comprehend the structure and organization of complex software to do maintenance or reverse engineering.

Software visualization techniques just mentioned are designed to address different cognitive needs during program comprehension. Mixed findings of existing studies led Storey et al. (2000) to the conclusion that a software visualization tool should provide support for different comprehension strategies such as top-down, bottom-up, or as needed. One type of visualization is inadequate for catering to all software comprehension needs (Maletic, Marcus, & Collard, 2002). Such tools should not only support multiple comprehension strategies but also

allow seamless transitions between comprehension strategies (Storey et al., 2000).

Existing studies on the relationship between software visualization and comprehension report mixed findings. Most studies rely on quantitative performance measures, postexperiment questionnaires, and interviews with stakeholders (Dunsmore, Roper, & Wood, 2000), and thus do not reveal the moment-by-moment details of the process through which programmers interact with visualizations in practice. In addition to this, the task context and the complexity of the software considered vary across studies, and only a small number studies focus on software maintenance tasks performed at a realistic scale. Therefore, there is a need for systematic studies of the microlevel details to reveal in what practical ways software visualization contributes to software comprehension. In an effort to address this need, we conducted a controlled eye-tracking study that combined quantitative and qualitative methods to investigate how programmers made use of visualizations during a realistic software maintenance task.

3. MATERIALS AND METHODS

3.1. Experimental Design

Thirteen software engineers who have on average 10 years of software development experience were recruited for this study. Proctor, Vu, and Salvendy (2002) argued that novices and experts are distinguished in terms of the knowledge structures or mental models they possess in a domain, which renders novices' reasoning rather shallow and less functional when compared to the experts. For this reason, participants who were similar in terms of their level of education and programming expertise were selected for this study. Subjects in the experiment group used the NDEPEND software visualization tool (described in the next section) to work on the program comprehension and maintenance tasks, whereas the control group relied on the standard graphical user interface provided by Visual Studio .NET to navigate the source code. Five participants were assigned to the control group, and eight subjects were assigned to the experiment group. While the participants were working on the software comprehension and maintenance tasks, their eye-gaze movements and screens were recorded with the Tobii T1750 eye-tracking system. Eye fixations were overlaid on the screen recordings by using the Tobi Studio eye-tracking analysis software.

Due to the constraints on subjects' availability and the necessity to conduct the study in a controlled lab setting, we opted for analyzing program comprehension behavior in a task environment, which is smaller in scale as compared to an industry grade software application. On the other hand, the software was complex enough to provide a realistic setting for the kinds of maintenance operations typically performed in the industry. To set the task environment, we developed an e-commerce application (~1000 LOC) in Microsoft ASP.NET that is designed to manage the product inventory and orders received by a small-scale business enterprise. The application reflects typical features of most e-commerce applications such as displaying a number of products in the inventory, querying for products in the database, adding selected products to an order list, and processing purchases. Thus, the experimental application has a modular complexity comparable to a real-world web application, which allowed us to design realistic program comprehension and maintenance tasks that can be studied in an experimental setting.

During the experiment, subjects were asked to perform seven programming tasks on our e-commerce application. Participants in the experiment group were provided a brief tutorial about the main features of the NDEPEND visualization tool before the experiment, as none of them had used this tool before. No training was provided to the control group members, because they were already expert users of the Visual Studio .NET integrated development environment (IDE). Subsequently, the participants were asked to complete the seven experimental tasks described next:

- Task 1 (3 min): Find the code portion in which the customer list is filled.
- Task 2 (3 min): Find the code portion in which the product list is filled and update the code for the list to display six products instead of five by default.
- Task 3 (4 min): In the products page, when the users enter the name of a product and press the "search" button, the website returns only the exact search results. However, they should also see those products whose names partially match the search term. Find the program fragment related to this case and update the code so that partial matches are also displayed.
- Task 4 (4 min): When searching for a customer record, users can use the name and area properties only. Add a new element to the search form, so that users can also search by last name.
- Task 5 (4 min): The "phone" information of the customer will not be required in the registration form anymore. Do the required changes for this purpose.
- Task 6 (7 min): Execute the program and follow the instructions below.

  - Log in to the system.
  - Go to the "orders" page.
  - Press the "Select Customer" button.
  - Select a customer from the customer list.
  - Press the "Add Item" button.
  - Select the product having a price of 18,09.
  - Set the quantity of the products that will be purchased as 2.
  - Press the "Continue Order" button.
  - See the total price field. You should see 36 instead of "36,18."

- You explored a bug which is a computation error in the fractional part of the price. Locate the bug in the source code and fix it accordingly.

- Task 7 (3 min): Find the program portion in which the product picture is read from the database and loaded to the screen.

To control the total duration of the experiment and the uniformity of the scale of measurements, a time out was set for every task by taking into account each task's difficulty and its estimated duration. Each experimental session lasted about 1 hr. The participants in the visualization group received a brief tutorial on the basic features of NDEPEND and how it can be used in conjunction with Visual Studio before the experiment.

Similar to Sim and Storey's (2000) study, we aimed to design tasks that are representative of a range of tasks a web developer would face in his or her daily work. For example, in our experiment participants were asked to repair a defect/bug or to add a new feature, rather than simply performing data flow analysis. Consequently, carrying out these tasks required participants to develop an understanding of the source code, which provided us a perspicuous setting to observe the impact of software visualization tools on software comprehension. After the experiment, a short semistructured interview was administered where participants were asked about their opinions regarding the difficulties they experienced with the tool and the tasks.

3.2. NDEPEND Software Visualization Tool

In this study a commercially available software visualization tool named NDEPEND is used, which includes several visualization techniques for representing Microsoft Visual Studio.NET programs. NDEPEND is specifically designed to help programmers analyze and explore software written in the Visual Studio.NET framework. Using NDEPEND, software developers can

- Evaluate the relative size and complexity of code segments based on various code metrics (e.g. lines of codes, cyclomatic complexity, number of children, etc.) and levels (methods, types, etc.) with a treemap that implements the block metaphor (Figure 1).
- Track program flow by using graphical demonstrations of dependencies in the code (Figure 2).
- Demonstrate the complex call relationships of the program with the matrix view (Figure 3).
- Compute customized search queries and code metrics with a special query language (Code Query Language-CQL).

During the experiments subjects in the experiment group particularly focused on the treemap and program flow visualizations provided by NDEPEND. Participants also had access to the Visual Studio environment because maintenance tasks required subjects to do some changes in the source code and test the updated application.

3.3. Visual Studio.NET as a Standard Software Development Environment

The control group used the standard Visual Studio.NET 2008 IDE to navigate through the source code. Visual Studio.NET offers developers an interface called "solution explorer," which provides a tree-based hierarchical representation of the folder organization within the source code (Figure 4). Users can inspect the organization of the source code by navigating through the folder hierarchy and read the contents of individual files across multiple tabs. The tabs provide a color-coded view of the source code together with hyperlinked class references. Subjects in the control group relied on these resources and the default debugger to work on the maintenance tasks.

3.4. Grounded Theory Methodology

In addition to quantitative analysis of outcome measures, we employed a grounded theory (Glaser & Strauss, 1967) approach to perform a qualitative analysis of the screen recordings collected during the experiment. Overall, we employed a mixed methodology in an effort to better capture and describe the patterns of use evidenced in the video recordings of both groups of participants. Qualitative research is directed primarily at collecting and analyzing nonnumeric data with the aim of achieving information depth rather than breadth (Coleman

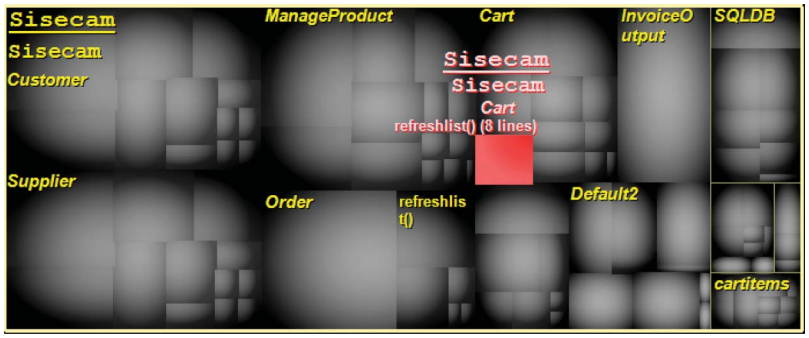

FIG. 1. A block visualization of the source code (color figure available online).

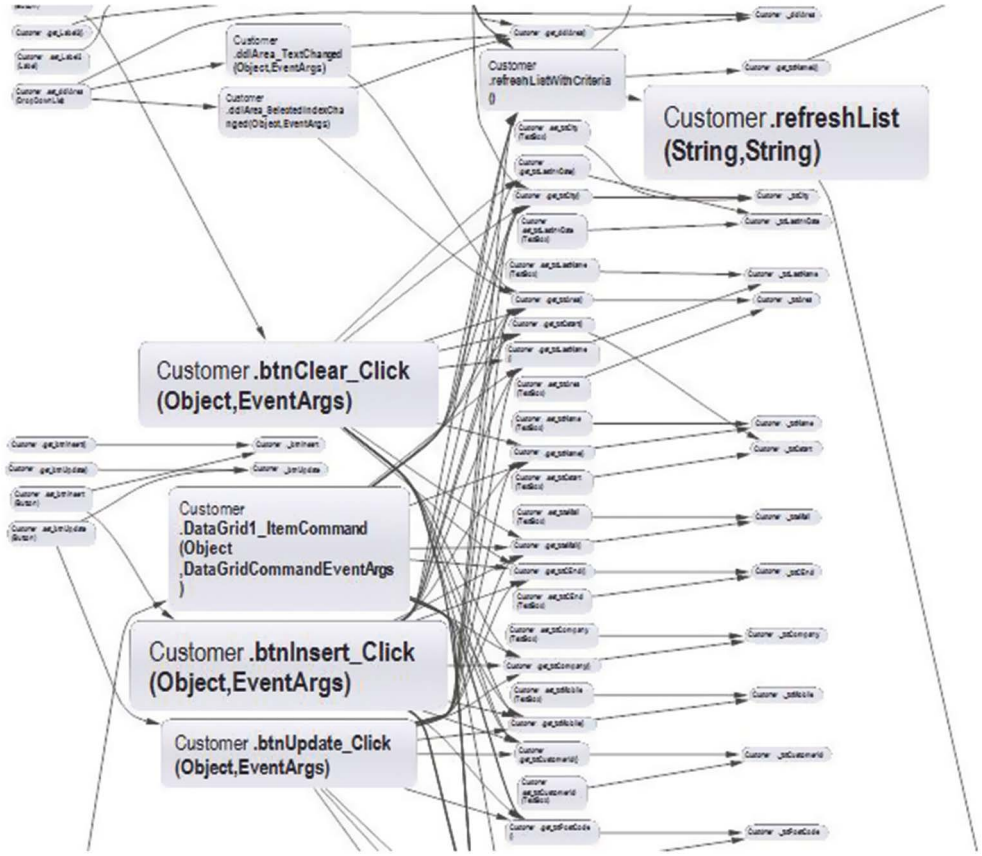

FIG. 2. Code flow and dependencies among modules are captured by the graph visualization (color figure available online).

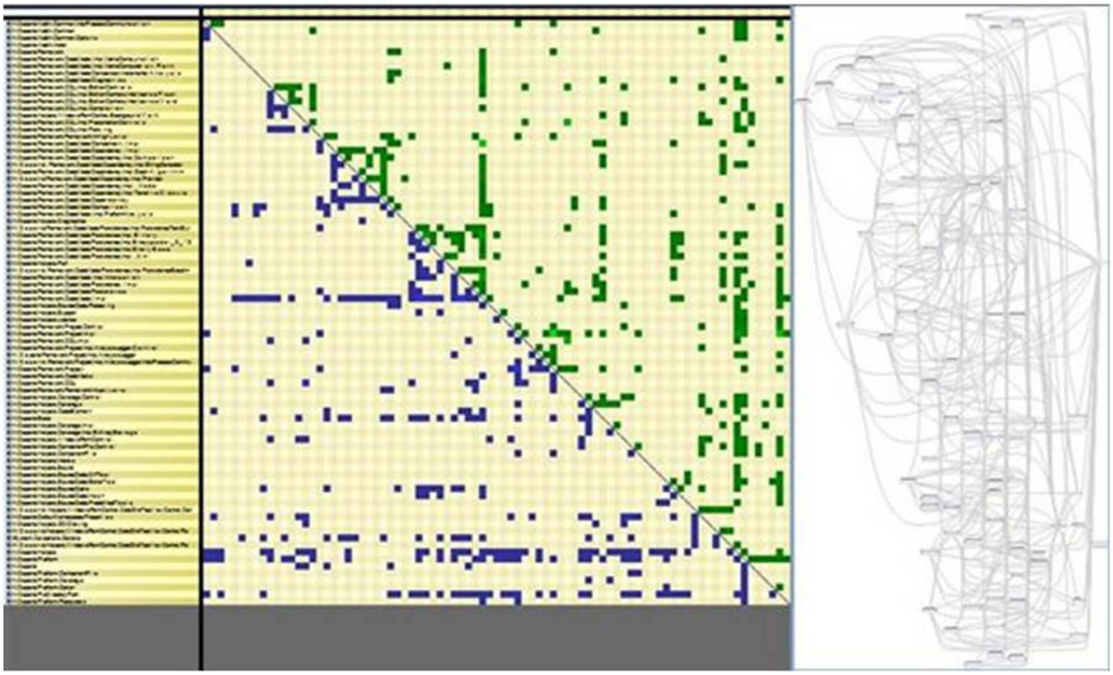

FIG. 3. Matrix visualization of the call relationships (color figure available online).

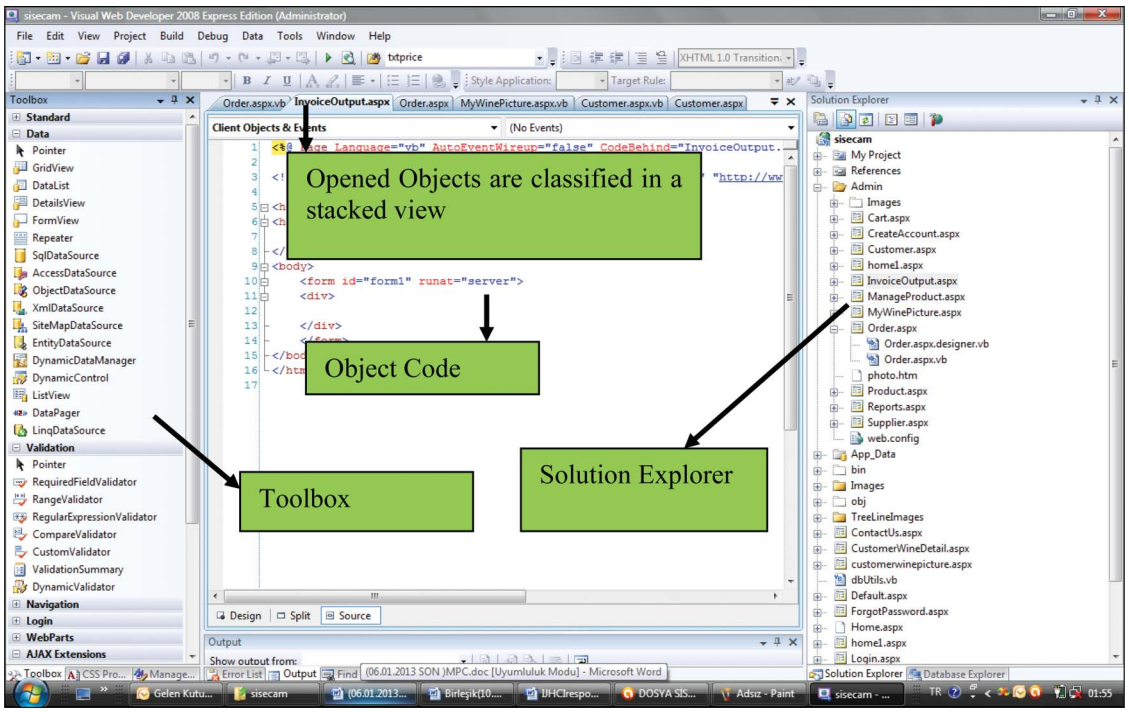

FIG. 4. Visual Studio.NET 2008 Integrated Development Environment. The solution explorer is located on the right column (color figure available online).

& O'Connor, 2007). Thus, as a qualitative research paradigm, the grounded theory approach is not particularly concerned with the statistical modeling of the data, but it focuses on uncovering how the observed activity is structured and organized through systematic coding of actions at various levels of granularity.

Grounded theory methodology has been employed in information systems research to study various aspects of system design and use, particularly to conduct ethnographic studies of the work setting in which software engineers develop software. Existing studies focus on various aspects of software development practice, such as the influence of software testing and communication practices on the development process and the quality of the product (Taipale & Smolander, 2006), when and why software process improvement practices are employed in the work setting (Coleman & O'Connor, 2007), what type of questions programmers raise when they evolve a code base, how are those questions related to the specific problem context, and what tools they use to address them (Sillito, Murphy, & de Volder, 2006).

Information visualization is another domain relevant to this study in which grounded theory methodology has been fruitfully employed. Such studies focus on how visualizations of data are used to coordinate decision-making processes in the context of building design (Tory & French, 2008) and sales forecasting (Asimakopoulos, Fildes, & Dix, 2009). Grounded theory methodology is also employed for developing a broader evaluation framework to conduct contextual analysis of information visualization systems (Zuk, Collins, & Carpendale, 2008). Such studies provide in-depth insights regarding how visualizations contribute to the work practices in the respective domains. Nevertheless, in none of the aforementioned studies grounded theory has been employed specifically to study how software visualization techniques mediate software comprehension strategies of programmers. Moreover, these studies primarily rely upon field observations and interviews as primary data sources. In this study we aim to build on existing applications of grounded theory in the information systems and usability literature by conducting a grounded theory analysis of video recordings overlaid with the participants' eye fixations.

3.5. Data Analysis

Both quantitative and qualitative methods were employed for the analysis of collected data. The experimental groups were compared in terms of performance outcome measures such as accuracy and task completion times to observe if the visualization had a positive effect on software maintenance performance. IBM SPSS version 19 was used to run the statistical analysis. In addition to this, a grounded theory approach was employed to identify in what specific ways visualizations contribute to the programmers' task performance, based on a close analysis of the participants' actions and eye recordings. Qualitative analysis aims to elaborate on the patterns indicated by our quantitative findings.

3.6. Quantitative Analysis Results

A two-way mixed analysis of variance (ANOVA) performed on task accuracy, where task and group are the within- and between-subjects variables, respectively, found a significant main effect of experimental condition, $F(1, 11) = 11.638$, $p < .01$, $\eta^2 = 0.514$, and a significant main effect of task, $F(6, 66) = 4.619$, $p < .01$, $\eta^2 = 0.296$. The means plot in Figure 5 indicates that the visualization group had higher accuracy percentage as compared to the control group across all tasks. The combined accuracy of the experiment group over all tasks was 85%, whereas the control group had a combined accuracy value of %46.

The means plot in Figure 5 suggests that participants' performance in both groups had improved as they progressed through the maintenance tasks in time. In other words, as the participants in both groups become familiar with the organization of the source code, the task accuracy has increased. The only exception to this pattern is Task 4, which has the lowest average accuracy percentage as compared to other tasks. Sidak corrected post hoc tests found that the accuracy percentage observed in Task 4 is significantly lower than Tasks 6 (M difference $= -0.525$, $p < .05$) and 7 (M difference $= -0.525$, $p < .05$). Task 4 turned out to be particularly difficult for the participants in the control group, as no one could complete the task correctly in the allotted time. The decrease in the performance of the visualization group on Task 4 was much smaller.

The means plot in Figure 5 also shows how the learning curves associated with both groups differ from each other. Participants in the experimental group quickly developed an understanding of the source code and reached perfect accuracy toward the end of the experiment. The improvement in the control group was relatively slower. The interaction effect of group and task was not significant, $F(6, 66) = 1.258$, $p > .05$, which indicates that the participants in the visualization group consistently performed better than the participants in the control group across all tasks.

None of the participants were familiar with the visualization tools provided by NDEPEND, so one might expect to observe longer task completion times because subjects had to make sense of these new forms of representations in relation to the source code. To observe the influence of software visualization on task completion times, a two-way mixed ANOVA was conducted with group and task as the between- and within-subjects variables, respectively. We only considered successfully completed tasks and excluded those tasks where less than 20% of the subjects could complete the task.

The ANOVA results indicated that there is a significant main effect of task type on task completion times, $F(3, 18) = 5.962$, $p < .01$, $\eta^2 = 0.498$. Planned repeated contrasts between levels of tasks indicated that subjects spent on average significantly more time on Task 5 as compared to 6, $F(1, 6) = 11.762$, $p < .05$, $\eta^2 = 0.662$. Planned repeated contrasts also found a significant difference between Tasks 6 and 7, $F(1, 6) = 8.462$, $p < .05$, $\eta^2 = 0.585$. Hence, some of the tasks required participants to spend significantly more amount of time to succeed (see Figure 6).

The effect of group on task completion time was only marginally significant, $F(1, 6) = 5.799$, $p = .053$, $\eta^2 = .49$, whereas the interaction effect was not significant, $F(3, 18) = 1.602$, $p > .05$. The means plot in Figure 6 suggests that participants in the control group took more time to complete Task 3 as compared to the experimental group. Task completion times for Tasks 5 and 6 were similar across both groups. Finally, the control group outperformed the visualization group in the last task.

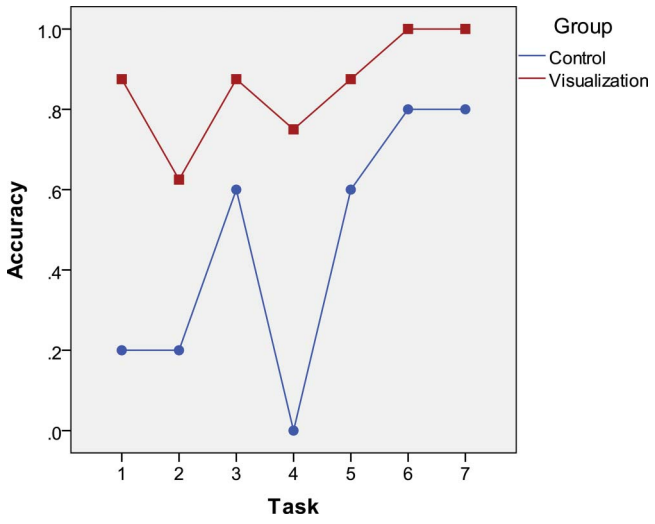

FIG. 5. Means plot for accuracy percentage across experimental conditions and software maintenance tasks (color figure available online).

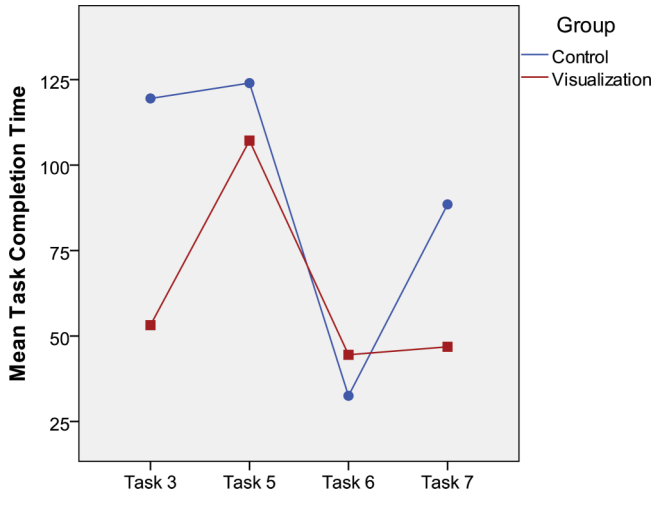

FIG. 6. Means plot for task completion times across tasks and experimental groups. Note. Only those tasks where accuracy percentage was higher than 20% were considered for comparison (color figure available online).

3.7. Qualitative Analysis Results

A grounded theory approach (Glaser & Strauss, 1967) was employed to perform a close analysis of the video recordings of participants' mouse gestures, typing actions and eye movements to identify in what specific ways visualizations contributed to programmers' task performance. Eye-tracking data was overlaid over the videos to aid the interpretative analysis of each subject's activity during maintenance tasks. Extended fixation durations were considered as indication of sustained attention to the vicinity of the objects/graphs on that part of the screen. Because the interface was dynamically changing while subjects were navigating over the visualizations through the course of each task, standard techniques such as areas of interest or heat-map analyses were not conducted on the eye-tracking data. In particular, our analysis aimed to identify and categorize the main software comprehension strategies evidenced in the actions of the participants. The analysis covered a total of $7 \times 13 = 91$ video excerpts, which amounts to a total duration of 10 hr of footage.

Initially, the software comprehension behaviors of the participants were interpreted to form primitive categories. Table 1 shows a portion of this *pre open coding* study for Task 4 of two arbitrarily selected participants. The verbal descriptions of user actions in both interface conditions aim to capture what features users oriented to as resources for action, how they enacted those resources as part of their search for a solution to the task at hand, and how they reacted to the consequences of their actions. Hence, these descriptions resemble Clarke's (2005) notion of "situational maps," which aim to identify a space of relevant concepts manifested in the data and their semantic relationships depending on how those concepts are enacted by the participants. However, our approach to grounded theory also differs from Clarke's approach in an important way, because our aim is to capture the sequential organization of actions evidenced in the course of participant's orderly activity. In the subsequent stages of our qualitative study, the categories identified in this manner will be linked to each other to uncover the orderliness of activities across both user groups.

After that, the video excerpts were first transcribed into a *narrative form* that summarized the main problem-solving steps and described where subjects allocated their attention on the interface. In particular, we focused on how participants determined the starting point of their analysis, how they found the relevant code context, how they reasoned about what needs to be fixed, and what kind of fixes they performed. For instance, the following paragraphs exemplify the narrative summaries of the performance of two subjects from each group during Task 4, which requires participants to add a new text area to an existing web page to support search by surname.

Experiment group, Participant 5, Task 4. The subject initially inspects the block visualization of the source code provided by NPDEND. In this screen he fixates sequentially on the objects from the top-left corner to the bottom-right. After getting an overall impression of the source code, he orients to the dependency graph visualization to follow the program logic. Once he locates the representation for the "Customer" object, he makes use of the graph to observe the internal code flow within the methods of the object. After this point, the subject switches to the Visual Studio.NET IDE and attempts to match the visual representations with the program code. He first finds the customer object which he examined before in NDEPEND, then through a systematic investigation of this context he reaches the "refreshListWithCriteria" method that contains the target code context to be maintained. Finally, he makes use of the existing field definitions to add the desired field and successfully completes the task.

Experiment group, Participant 3, Task 4. Participant executes the program to make sense of the flow logic of the related web page that is required to be maintained. He monitors the visualization based on his inspection of the program execution process. Once the subject identifies the relevant objects with the help of the visualization, he orients to the Solution Explorer to find where they are defined in the source code. He employs a systematic understanding strategy to locate the relevant code context to be maintained. Nevertheless, the subject eventually gets lost in the source code and runs out of time before he could do the desired modification to the code.

Control group, Participant 5, Task 4. After a brief investigation on the Solution Explorer, the participant runs the program to follow the execution steps for the targeted part of the software. He identifies "customer.aspx" as the page that needs to be modified. Then he moves his attention to the solution explorer window and browses through the class tree to identify objects that are semantically related to the given task. The subject inspects the names of the objects to locate the relevant code segments to be maintained but fails to locate the target context in the time allotted.

Control group, Participant 2, Task 4. Participant first executes the program and navigates to the "customer.aspx" page through the main menu of webpage. Once he realizes that this is the page that needs to be modified to add the desired feature, he switches to the Visual Studio Solution Explorer IDE and searches for the relevant code context. But he fails to develop a strategy to find the relevant object that populates this webpage.

Next, the narrative descriptions just exemplified were further segmented and categorized through *open coding*, which aimed to capture and label recurrent themes/actions observed across all sessions. Some of the actions interpreted and identified at this stage include switching to the code view, tracking the code flow, matching visualizations with the source code, fixating on specific code segments like an SQL statement or a mathematical expression, following the program flow, searching for a desired code fragment with the search utilities of the IDE, getting lost in the code, and succeeding or failing the task.

In the *axial coding* stage of our analysis, openly coded software comprehension actions for all tasks were consolidated and generalized under more abstract descriptive categories in an

TABLE 1
Software Comprehension Behavior Categories Observed During All Tasks

Participant 5 (Experiment Group)
Task 4: When searching for a customer record, users can use the name and area properties only. Add a new element to the search form, so that users can also search by last name.

Time	User Action	System Response	Candidate for Categorization
00:00–00:20	The subject initially starts to inspect the block visualization of the source code provided by NPDEND.	NDEPEND application opened and block visualization screen is displayed.	Try to match visual representations with the program code
00:20–00:27 00:27–00:30 00:30–01:09	He fixates sequentially on the objects from the top-left corner to the bottom-right.		
01:09–01:14	After getting an overall impression of the source code, he orients to the dependency graph visualization to follow the program logic.	System shows the "dependency graph" for the program.	
01:14–02:07	He inspects the graph and locates the representation for the "Customer" object		
02:07–02:07	Once he locates the representation for the "Customer" object, he makes use of the graph to observe the internal code flow within the methods of the object.	System shows the "dependency graph" of the methods within the customer object.	
02:07–02:13	He clicks on the zoom feature to have a close view of the graph	System enlarges the dependency graph.	
02:13–04:30	He investigates internal methods of the object	System scrolls to the method which is under investigation	
04:30	After this point, the subject switches to the Visual Studio.NET IDE and attempts to match the visual representations with the program code.	Visual Studio.Net Environment is displayed on the screen	
04:30–10:43	First three tasks take place, and then the fourth task is introduced.		
10:43–10:55	Fixates over the Solution Explorer		
10:55	He finds the customer object which he examined before in NDEPEND.		
10:55	Double-clicks on the "Customer" object (Searches for the relevant context in which the solution may exist)	System switches to the code view of the "customer" object	Find the desired object on the solution explorer

(Continued)

TABLE 1
(Continued)

Participant 5 (Experiment Group)
Task 4: When searching for a customer record, users can use the name and area properties only. Add a new element to the search form, so that users can also search by last name.

Time	User Action	System Response	Candidate for Categorization
10:55–11:00	Investigates the program flow line by line in the customer object beginning from the Page_load event	System scrolls to the desired program portion	Follow the "Systematic Understanding" strategy to locate the relevant code context
11:00–11:10	Searches for the desired code in context by reading and scrolling down. Come to "refreshlist" program portion.		Follow the "Systematic Understanding" strategy to locate the relevant code context
11:10–11:35	Come to "refreshlistwithcriteria" program portion		Use "Systematic Understanding" strategy to find the desired code line
11:35–13:40	Finally, he makes use of the existing field definitions to add the desired field and successfully completes the task.	System compiles and runs with success	Success

Participant 2 (Control group)
Task 4: When searching for a customer record, users can use the name and area properties only. Add a new element to the search form, so that users can also search by last name.

Time	User Action	System Response	Candidate for Categorization
00:00–12:02	First Three tasks take place, and then the fourth task is introduced.		
12:02	Clicks on Visual Studio.Net	The System opens the Visual Studio.Net Environment	
12:02–12:36	Explores the diagram by hovering the mouse over different program entities.	System shows the name of the object while the user hovers the mouse over a node on the graph.	
12:36–13:41	The participant runs the program to follow the execution steps for the targeted part of the software.	System runs and shows the customer search screen. Then shows the result list according to given search criteria	Runs the program and inspects web pages to identify relevant objects

(Continued)

TABLE 1
(Continued)

Participant 5 (Experiment Group)
Task 4: When searching for a customer record, users can use the name and area properties only. Add a new element to the search form, so that users can also search by last name.

Time	User Action	System Response	Candidate for Categorization
13:41–14:00	He identifies "customer.aspx" as the page that needs to be modified.		Investigate the solution explorer to locate relevant objects. Object names are the only clue (Appropriate naming is mostly important)
14:00–14:18	Explores the diagram again by hovering the mouse over different program entities.	System shows the name of the object while the user hovers the mouse over a node on the graph	Couldn't develop a specific comprehension strategy
14:18	Admit failure		Fail

Note. Categories related to the experimental and control group are marked differently.

effort to capture the variety of software comprehension related actions observed in both experiment groups. The categories considered for each group that were relevant across all tasks are displayed in Table 2. These categories allowed us to prepare a summary diagram of each task for each experimental group.

Next, we applied these codes to the narrative summaries to produce a *sequential representation* of the actions performed by the subjects during each task. Tables 3 and 4 display the coded sequences corresponding to the performance of each participant in all tasks in the control and experimental groups, respectively. The distribution of software understanding strategies employed by each group in response to specific tasks can be observed from these tables. The orderliness in the actions of the subjects in the visualization group allowed us to further organize their code sequences into three larger segments, as indicated by the three columns in Table 4. Codes that refer to software comprehension strategies such as bottom-up, systematic, and as needed are based on the definitions proposed by related work in the software comprehension literature.

The next stage of our qualitative analysis involved producing *axial coding graphs*, which combine the separate activity sequences identified for all participants for each task. This resulted in seven axial coding graphs which summarized the collective performance of each group over each task. For instance, Figures 7 and 8 display the axial coding diagrams of the experiment and control groups respectively for Task 4. The figures represent the program understanding and program maintenance behaviors exhibited by the participants with boxes. The color-coded connections between boxes reveal how each participant moved from one category of action to the other during the corresponding task. Axial coding graphs aim to make it visible what software comprehension patterns were exhibited by experiment and control groups during each task, and to what extent those patterns were similar or different across both groups.

At the final stage of our qualitative analysis, we performed a *selective coding* over the axial coding graphs to produce an overall summary of core categories of the program comprehension behaviors exhibited by each group. Selective coding resulted in a single flowchart for the experiment and control groups, which are displayed in Figures 9 and 10, respectively. Selective coding graphs provide an abstraction over task-specific axial coding graphs, which display the specific software comprehension patterns collectively exhibited by each group of participants during the entire experiment.

4. DISCUSSION

Our quantitative analysis of outcome measures indicates that the participants in the visualization group had a higher accuracy percentage across all tasks. Moreover, although both groups improved their performance as they become acquainted with the source code, the participants in the experimental condition exhibited a steeper learning curve. The difference between the two groups' performance was particularly different in Tasks 1, 2, and 4. However, it is not easy to see from the outcome measures how visualizations contributed to this difference in practice. When we compared task completion times in the successful cases, processing the visualizations did not seem to

TABLE 2
Software Comprehension Behavior Categories Observed During All Tasks

Code	Description of Observed Behavior
E1	Find the desired object on the solution explorer
E2	Find the desired object on the NDEPEND interface
E3	Try to match visual representations with the program code
E4	Follow the "Systematic Understanding" strategy to locate the relevant code context
E5	Use "Bottom-up" strategy to find the relevant code context (The write order of the methods are important)
E6	Use "As Needed" strategy with some hypothesis to find the context
E7	Lead into a wrong path, but quickly bactracks from it
E8	Use "Systematic Understanding" strategy to find the desired code line
E9	Continue using systemmatic strategy and find the context
E10	Fail to develop a strategy to find the desired code line
E11	Use the "Bottom-up" strategy to search for the code line
E12	Success
E13	Fails to complete the task (timeout)
E14	Execute the program and try to understand the flow logic
E15	Search the desired object systemmaticaly on the NDEPEND interface
E16	Use bottom-up strategy to find the relevant code context
E18	Lost in the code context
E19	Use as needed strategy with the help of standard SQL/PL tokens as search keywords
E20	Follow the program flow in NDEPEND
E21	Use mathematical statements as a search phrase to find the desired code segment
C1	Investigate the solution explorer to locate relevant objects. Object names are the only clue (Appropriate naming is mostly important)
C2	Inspect the solution explorer (evidenced by extended fixation durations)
C3	Couldn't develop a specific comprehension strategy
C4	Find the context with the bottom-up strategy
C5	Use as needed strategy to follow an inaccurate hypothesis
C6	Lost in the source code
C7	Follow the code flow using systematic software comprehension
C8	Fails to complete the task
C9	Success
C10	Runs the program and inspects web pages to identify relevant objects
C11	Use top down comprehension strategy to explore an irrelevant object due to its misleading name
C12	Use as-needed strategy with some hypothesis and find the context
C13	Find the desired object in the solution explorer in a short time
C14	Use bottom-up strategy to find the context
C15	Find the desired code fragment with the bottom up strategy
C16	Come to realize that he/she is headed in the wrong way (without developing a specific comprehension strategy)
C17	Use Systematic Comprehension strategy to find the desired code segment
C18	Take advantage of standard SQL/PL keywords and find the desired code segment
C19	Investigate the method definitions sequentially with bottom up strategy
C20	Investigate the method definitions with as-needed strategy
C21	Find the desired mathematical expression with as-needed strategy
C22	Devises a correct hypothesis that eventually lead him/her to the desired expression via an as-needed strategy

Note. Categories related to the experimental and control group are marked differently.

TABLE 3
Coding of Each Participant's Software Comprehension Actions in the Control Group for Each Task

Participant No.	Software Comprehension Behaviors
Control Group Task 1	
1	C1-C4-C7-Success
2	C1-C2-C3-C5-C6- Fail
3	C1-C2-C3-C5-C6- Fail
4	C1-C2-C3-C5-C6- Fail
5	C1-C2-C3-C5-C6-Fail
Control Group Task 2	
1	C1-C13-C6-Fail
2	C1-C11-C5-C16-Fail
3	C1-C11-C12-C15-Success
4	C1-C11-C5-Fail
5	C1-C11-C14-C5-C16-Fail
Control Group Task 3	
1	C10-C1-C14-C18-Success
2	C10-C1-C3-Fail
3	C10-C1-C5-Fail
4	C10-C1-C14-C18-Success
5	C10-C1-C16-C17-C18-Success
Control Group Task 4	
1	C1-C13-C7-C19-C6-Fail
2	C10-C1-C3-Fail
3	C1-C12-Fail
4	C1-C13-C7-C19-C6- Fail
5	C10-C1-C13-C12-Fail
Control Group Task 5	
1	C10-C1-C13-C3-Fail
2	C10-C1-C16-C3-Fail
3	C1-C16-C7-C15-Success
4	C1-C16-C15-Success
5	C10-C1-C16-C7-C15-Success
Control Group Task 6	
1	C1-C12-C21-Success
2	C1-C5-C3-Fail
3	C10-C1-C12-C21-Success
4	C10-C1-C12-C21-Success
5	C10-C1-C12-C21-Success
Control Group Task 7	
1	C10-C1-C16-C22-Success
2	C1-C14-C5-Fail
3	C1-C16-C22-Success
4	C10-C1-C12-C7-Success
5	C10-C1-C12-C7-Success

TABLE 4
Coding of Each Participant's Software Comprehension Actions in the Experiment Group for Each Task

	Software Comprehension Behaviors		
Participant No.	Stage 1 : Comprehension of the organization of the software	Stage 2 : Identification of relevant context	Stage 3: Solution Finding
Experiment group Task 1			
1	E1-	E4	E8-Success
2	E2-E3-	E4	E8-Success
3	E1-E3-	E4	E8-Success
4	E1	E4	E8-Success
5	E2-E3-	E4	E8-Success
6	E1-		E5-Fail
7	E2-E3-	E4	E8-Success
8	E1-	E4	E8-Success
Experiment group Task 2			
1	E1	E4-E9	E8-Success
2	E2	E4-E7-E9	E10-Fail
3		E5-E9	E10-Fail
4	E1-E3	E6	E8-Success
5	E3	E4-E9	E10-Fail
6	E3	E4-E7-E9-E11	Success
7	E3	E4-E7-E9	E8-Success
8	E1-E3	E4-E9	E8-Success
Experiment group Task 3			
1	E14-E1		E19-Success
2	E14-E15-	E4-	E19-Success
3	E14-E1-	E4	Success
4	E14-E1-	E4-	E19-Success
5	E14-E1-E16	E7-E15	E18-Fail
6	E14-E15	E4-	E19-Success
7	E1	E4-	E19-Success
8	E14-E1	E4-	E19-Success
Experiment group Task 4			
1	E14-E1	E4	E8-Success
2	E14-E1	E4	E8-Success
3	E14-E1	E4	E8-E18-Fail
4	E14-E1	E4	E8-Success
5	E3-E1	E4	E8-Success
6	E14-E1	E4	E8-E18-Fail
7	E3-E1	E4	E8-Success
8	E14-E1	E4	E8-Success
Experiment group Task 5			
1	E3-E1	E4	E19-Success
2	E14-E1	E4	E19-Success
3	E14-E1	E4	E18-Fail

(Continued)

TABLE 4
(Continued)

Participant No.	Software Comprehension Behaviors		
	Stage 1 : Comprehension of the organization of the software	Stage 2 : Identification of relevant context	Stage 3: Solution Finding
4	E3-E1	E4	E19-Success
5	E3-E1	E4	E8-Success
6	E3-E1-	E4-	E19-Success
7	E3-E1	E4	E8-Success
8	E3-E1	E4	E19-Success
Experiment group Task 6			
1	E14- E	E4	E21-Success
2	E14- E1	E4	E21-Success
3	E14- E1	E4-	E19-E21-Success
4	E14- E1	E4	E19-E21-Success
5	E14- E1	E6	E19-E21-Success
6	E14-E3-E1-	E4	E19-E21-Success
7	E14- E1	E4	E19-E21-Success
8	E14- E1	E4	E21-Success
Experiment group Task 7			
1	E1- -	E6-E8	Success
2	E20-E3-	E4-E6-E8	Success
3	E14-E3-	E4-E8	Success
4	E20-E3-	E4-E6-E8	Success
5	E20-E3-	E4-E8	Success
6	E14-E3-	E4-E8	Success
7	E1	E6-E8	Success
8	E1	E4-E8	Success

significantly extend the time it takes to perform any of the tasks but the last one. Some of the tasks turned out to be more difficult than others. For instance, none of the participants in the control group could complete Task 4, whereas 70% of the participants in the visualization group could complete that task within the allocated time limit. It is not obvious from the definition of the task why such a difference was observed between both groups and to what extent the difference can be attributed to the presence of visualization resources. In an effort to address such questions, we expanded our quantitative analysis of outcome measures with a qualitative analysis of screen recordings and eye-gaze patterns of the participants.

By employing a grounded theory approach, we specifically aimed to uncover what kind of software comprehension strategies both group members employed as they were working on the tasks. We used the coding distributions summarized in Tables 3 and 4 and the axial codings for each group to better explain the statistical differences observed in Tasks 1, 2 and 4.

In Task 1, participants were asked to find the part of the source code where the list of customers was populated. Participants in the control group aimed to address this task by employing a bottom-up strategy via the solution explorer window, as indicated by the appearance of codes C1, C2, and C5. Only one participant was able to locate the correct code context in the control group. This participant was able to successfully complete the task by systematically reading the relevant code context. The remaining participants in the control group ended up following inaccurate hypotheses about what the relevant

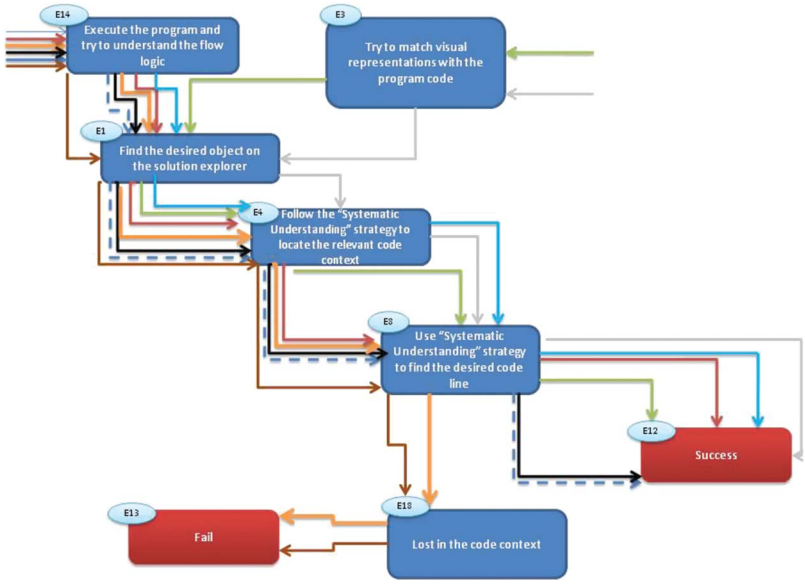

FIG. 7. Axial coding for the combined performance of the experiment group on Task 4. (color figure available online).

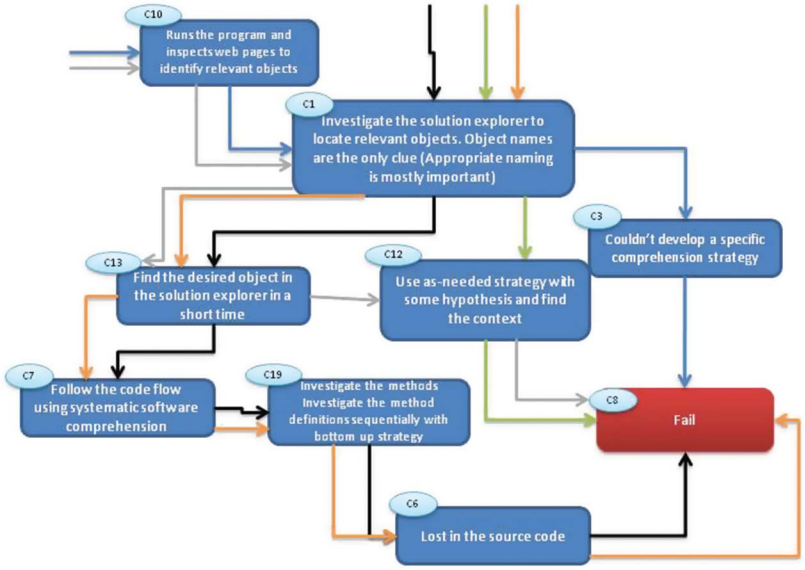

FIG. 8. Axial coding for the combined performance of the control group on Task 4. (color figure available online).

code context is (code C5) and switched to the "as needed" strategy to follow up on their guess. On the other hand, participants in the experiment group exhibited two different approaches to this task. Four participants initially attempted to match the visual representations with the relevant parts of the source code (codes E2 and E3). With the help of the visualizations, these participants ended up following a more systematic approach as they identified and explored the relevant objects, which eventually led them to the targeted code line. The remaining subjects in the experiment group relied on the more familiar solution explorer to browse through the source code (indicated by E1). They made use of the visualizations while they were systematically exploring the relationships between the relevant objects (indicated by E4). Three of the four subjects who exhibited this problem-solving behavior also succeeded in finding the target code line.

In Task 2, participants were asked to find the code context where the product list is filled and update the code so that six instead of five items would be displayed. Participants in the control group approached the task in a similar way as in Task 1, where they searched for potentially relevant objects in

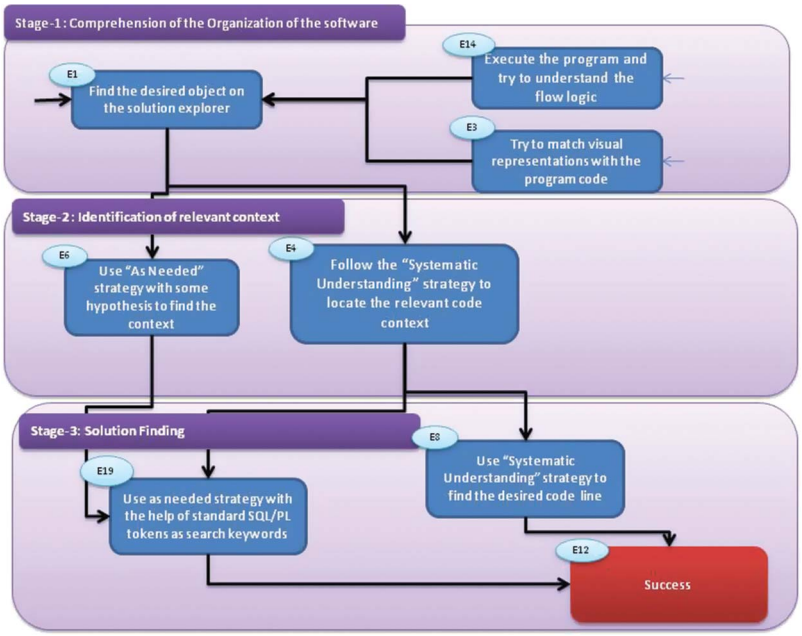

FIG. 9. Selective coding summarizing the experiment group's overall behavior (color figure available online).

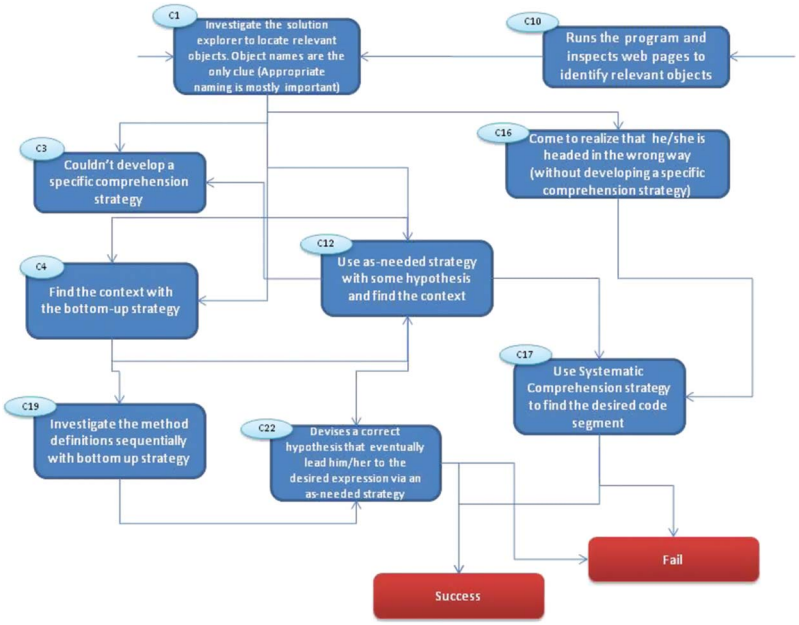

FIG. 10. Selective coding for the control group (color figure available online).

the source code with the solution explorer. Because the relevant object was named in a rather deceptive way, most subjects ended up exploring an irrelevant object (C11). Only one subject could backtrack and find the relevant object by employing the as-needed and then the bottom-up strategies. Participants in the visualization group were also initially misled by the deceptive object naming convention, but they were able to backtrack quickly due to the bird's-eye view perspective provided by the visualizations (E7). Thus, the visualizations turned out to be useful for identifying the relevant object in a systematic fashion. Most participants in the visualization group were able to find the relevant object, but three of them failed to find the particular code line in which the modification has to be made within the time allotted.

Task 4 involved the addition of a new search element to the page where the list of customers was searched. The summaries in Tables 3 and 4 as well as the axial coding of Task 4 (see Figures 7 and 8) for each group indicate that visualizations

turned out to be particularly useful to locate and systematically investigate the classes that need to be modified to accomplish this task. This task required participants to grasp how related classes work together to perform the search operation over products, so that they could modify them accordingly. Hence, it required participants to develop a deeper understanding of the source code as compared to other tasks. The control group had to rely on cues such as how relevant classes might have been named to locate them in the Solution Explorer (C1). Once they found a relevant class, they had to trace other relevant classes referenced from the code by sequentially reading the code and following the hyperlinks (C19). This led them to a bottom-up strategy, but due to the number of dependencies across implicated classes none of the participants in the control group could succeed. On the other hand, most participants in the experiment group started with executing the corresponding webpage to look for cues about implicated classes (E14 and E1). Two subjects directly started working on the visualizations and succeeded in finding the relevant objects (E3). Once candidate objects were identified, they made use of the dependency graph and the treemap to locate and inspect related classes altogether (E4). Overall the visualizations made it easy for the subjects in the experiment group to follow a systematic comprehension strategy.

It is also informative to consider the similarities and differences observed across both groups over those tasks that did not significantly distinguish the control group from the experiment group. For instance, in Task 3, subjects were asked to fix a bug in a search query that only returns exact matches. The desired search behavior should output those products that partially match the keywords provided by the user. Because the subjects were experienced developers, they quickly realized that the desired feature should include a change involved with a SQL statement that queries the product database. A majority of the subjects in both conditions were able to find the relevant code context by using the IDE's standard search feature with keywords such as "select," "from," and so on, that reflect the syntax used by such database calls (as captured by codes such as C18, E19).

Task 5 requires subjects to make a change in the registration page so that the phone information is no longer presented in the form. The selective codings for the control group reveal that participants resorted to bottom-up strategies for finding and analyzing the relevant code context. Some participants were misled into incorrect objects, but the majority were able to backtrack and find the appropriate code context. The visualization group continued to make use of the visualizations to aid their search for the relevant code context. Once they identify the context, some participants employed the as needed approach (E19), whereas others explored the code more systematically (E8). By this time participants in both groups had developed a broader understanding of the source code, which is evidenced in the way they navigate through the code to find the relevant code context.

Task 6 asked participants to fix a mathematical statement that computes the total cost of a given order, which produces incorrect results due to an incorrect casting of the variable that keeps the total value. The task walks the subjects through a few webpages to help them realize the problem. Control group members who succeeded in this task completed the task slightly faster than the visualization group members, possibly due to the cues provided by the walkthrough, which helped them locate the relevant code context. Then, they opted for the as needed strategy to locate where the problematic mathematical expression could be by using the text-search feature of the solution explorer (C12, C22). The visualization group again relied on the visualizations to navigate through the objects initially, but once they identified the context they exhibited a similar "as needed" approach by searching for specific expressions (E19, E21). The time they spent over the visualizations seemed to have contributed to the slightly higher task completion time as compared to the control group.

Finally, Task 7 requires subjects to find what part of the code queries the database for the product images. Most subjects in both groups exhibited an as needed approach because the task required a very specific location. Both groups seemed to have benefited from being familiar with the source code at this stage of the experiment. This may be the reason why they did not resort to searching for specific keywords as some participants did in Task 5.

The selective coding of visualization and control groups (see Figures 9 and 10, respectively) allows us to make a broader comparison between both groups in terms of their overall performance across all tasks. The experiment group's diagram identifies three main stages in their performance. The first stage can be named as "comprehending the organization of the software," where participants tried to comprehend the task and identify the objects relevant for the task. The experiment group did this by investigating the visualizations, executing the program, or following the program flow. The second stage can be named "identification of the relevant code context," where subjects located the relevant code context or object that needs to be modified to accomplish the task. The visualization group in general employed either systematic or as needed strategies at this stage. The last stage can be named as "solution finding" where participants followed either the systematic or as needed approaches to accomplish the task, that is, either show the targeted code snippet or implement the required maintenance work. In particular, the affordances of the visualizations for revealing the dependencies among relevant objects and code snippets allowed subjects in this group to develop a deeper understanding of the organization of the code, which is evidenced in their increased average accuracy as the experiment proceeded.

In contrast to the visualization group, the selective coding diagram of the control group does not reveal a similar degree of orderliness (see Figure 10). Left with default code-navigation resources provided by the IDE, participants in the control group

followed a variety of strategies to deal with each task. In particular, they tended to rely on the solution explorer to navigate the tree of object definitions and on running the program to identify which objects are relevant for a given maintenance task. The complexity of the source code and the lack of a bird's-eye view access to the dependencies within the source code led them to follow the "as needed" and "bottom-up" approaches in general. Some of the subjects tried to exploit the names of the classes, yet such semantic cues turned out to be often misleading due to naming conventions used in the source code. Thus, subjects tended to focus on accomplishing the specific fix itself rather than trying to understand the general organization of the source code. This approach often led to problems, especially for those tasks where there are no obvious clues such as specific SQL or math expressions. The control group's performance had also increased toward the end of the experiment. As they became more familiar with the organization of the source code, participants in the control group also began to employ more systematic strategies as opposed to bottom-up and as needed strategies.

5. CONCLUSION

Overall, our quantitative and qualitative findings indicate that the visualizations provided by NDEPEND contributed positively to the performance of the experiment group. In general, the visualizations guided participants to follow more systematic software comprehension strategies by allowing them to discover and trace the key dependencies between relevant objects and/or code snippets. As it is evidenced in their actions, guidance for systematic investigations allowed programmers in the experiment group to develop a better understanding of the source code as compared to the control group, which culminated into the statistically significant differences observed between both groups. Even though the sample size is rather small and the scope of the tasks is limited due to the constraints imposed by our controlled experimental setting, such differences highlight the significant influence of software visualization techniques on the software comprehension process.

However, our results do not simply suggest that providing mere access to a set of visualizations will eliminate all the challenges involved with software comprehension. Our qualitative analysis of video recordings highlighted that participants mainly use these visualizations as a resource for making sense of the organization of the source code. The visualizations were effective to the extent participants could tailor them to their specific needs for each task. During our experiments subjects often needed to switch between several software comprehension strategies while they were working on the tasks. Thus, coordinated use of both the conventional IDE and the visualizations emerged as a necessity. Most participants in the visualization group struggled to find effective ways to coordinate the content provided by the visualizations with the source code presented through the solution explorer interface of the IDE. This was largely due to the need for computing visualizations at different levels of granularity so that the relevant dependencies can be inspected at the appropriate level of analysis. Therefore, in an effort to make visualizations more effective and useful in the context of software maintenance, designers of software visualization tools may consider better ways to integrate these representations into standard IDEs to enable their coordinated use. Our findings suggest that the ability to seamlessly zoom in and out of selected parts of the source code visualizations may better support the programmers' needs for noticing and tracking the most important dependencies in the existing code. Such a level of flexibility may also provide better software support to the programmers by allowing them to easily switch between several software comprehension strategies based on their situated, dynamic information needs.

Finally, in this study we have also exemplified the use of grounded theory to conduct a qualitative analysis of eye-tracking data in the context of software comprehension. The grounded theory approach allowed us to capture how participants at each interface condition made use of the resources made available to them while they were working on software maintenance tasks. The categorizations grounded in users' actions and eye-gaze patterns allowed us to construct process models that revealed an important difference among the way the two groups organized their activities. The grounded categories enabled us to attribute this difference to the availability and purposeful use of visualizations. The combined use of grounded theory and process models in this manner may inform other usability studies that aim to go beyond black box measures by exploring how users interact and gradually make sense of system interfaces in specific task contexts.

REFERENCES

Asimakopoulos, S., Fildes, R., & Dix, A. (2009). *Forecasting software visualizations: an explorative study*. Paper presented at the the 23rd British HCI Group Annual Conference on People and Computers: Celebrating People and Technology, October 21, Swinton, UK

Bednarik, R., &Tukiainen, R. (2006). *An eye-tracking methodology for characterizing program comprehension processes*. Paper presented at the Eye Tracking Research & Applications Symposium March 27-29, San Diego, CA.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., . . . Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, *53*, 66–75.

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*, 543–554.

Coleman, G., & O'Connor, R. (2007). Using grounded theory to understand software process improvement: A study of Irish software product companies. *Information and Software Technology*, *49* 654–667.

Corritore, C. L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, *3*, 199–222.

Corritore, C. L., & Wiedenbeck, S. (2001). An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, *54*, 1–23.

Diehl, S. (2005). *Software visualization*. Paper presented at the ACM ICSE'05 27th International Conference on Software Engineering, May 15–21, St. Louis, MO.

Dunsmore, A., Roper, M., & Wood, M. (2000). The role of comprehension in software inspection. *Journal of Systems and Software*, *52*, 121–129.

Glaser, B., & Strauss, A. L. (1967). The discovery of grounded theory: Strategies for qualitative research. Chicago, IL: Aldine.

Grant, C. A. M. (1999). *Software visualization in Prolog*. Cambridge, UK: Queens College.

Jun, E., Landry, S., & Salvendy, G. (2011). A visual information processing model to characterize interactive visualization environments. *International Journal of Human-Computer Interaction*, *27*, 348–363.

Kagdi, H., Yusuf, S., & Maletic, J.I. (2007). *On using eye tracking in empirical assessment of software visualizations*. Paper presented at the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: WEASEL Tech 2007, November 5–9, Atlanta, GA.

Karahasanovic, A., Levine, A. K., & Thomas, R. (2007). Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *The Journal of Systems and Software 80*, 1541–1559.

Koenemann, J., & Robertson, S. (1991, April–May). *Expert problem solving strategies for program comprehension*. Paper presented at the ACM CHI 91 Human Factors in Computing Systems Conference, New York, NY.

Koschke, R., & Diehl, S. (2002). Software visualization for reverse engineering. *Software Visualization Lecture Notes in Computer Science*, *2269*, 524–527.

Lemieux, F., & Salois, M. (2006). Visualization techniques for program comprehension: A literature review. *The 2006 Conference on New Trends in Software Methodologies Tools and Techniques: Proceedings of the Fifth SoMeT06*, 22–47.

Littman, D. C., Pinto, J., Letovski, S., & Soloway, E. (1986). *Mental models and software maintenance*. In *Proceeding papers presented at the First Workshop on Empirical Studies of Programmers* (pp. 80–98). Norwood, NJ: Ablex.

Maletic, J. I., Marcus, A., & Collard, M. L. (2002). *A task oriented view of software visualization*. Paper presented at the IEEE First International Workshop on Visualizing Software for Understanding and Analysis, June 26, Paris France.

Petre, M. (2010). Mental imagery and software visualization in high-performance software development teams. *Journal of Visual Languages and Computing 21*, 171–183.

Price, B. A., Baecker, R. M., & Small, I. S. (1993). A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, *4*, 211–266.

Proctor, R. W., Vu, K.-P.L., & Salvendy, G. (2002). Content preparation and management for web design: Eliciting, structuring, searching, and displaying information. *International Journal of Human-Computer Interaction*, *14*, 25–92.

Shaft, T. M., & Vessey, I. (1995). The relevance of application domain knowledge: the case of computer program comprehension. *Information Systems Research*, *6*, 286–299.

Sillito, J., Murphy, G. C., & de Volder, K. (2006). *Questions programmers ask during software evolution tasks*. Paper presented at the SIGSOFT'06/FSE-14, Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering, November 5–11, Portland, OR.

Sim, S. E., & Storey, M. D. (2000). *A structured demonstration of program comprehension tools*. Paper presented at the Seventh Working Conference on Reverse Engineering, November, 23–25, Brisbane, Australia.

Soloway, E., Adelson, B., & Ehrlich, B. (1988). Knowledge and processes in the comprehension of computer programs. *The Nature of Expertise*, pp. 129–150.

Storey, M. A., Wong, K., & Muller, H. A. (2000). How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, *36*, 183–207.

Telea, A., Ersoy, O., & Voinea, L. (2010). *Visual analytics in software maintenance: Challenges and opportunities*. Paper presented at the International Symposium on Visual Analytics Science and Technology.

Umphressa, D., Hendrixa, T. D., Cross, J. H., & Maghsoodloob, S. (2006). Software visualizations for improving and measuring the comprehensibility of source code. *Science of Computer Programming*, *60*, 121–133.

Voinea, L., Telea, A., & Wijk, J. (2005). *CVSscan: Visualization of code evolution* Paper presented at the ACM Symposium on Software Visualization (Softviz 2005), May 14–15, Saint Louis, MO.

von Mayrhauser, A., & Vans, A. (1996). Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, *22*, 424–437.

Wettel, R., & Lanza, M. (2008). *Codecity 3D visualization of large-scale software*. ICSE'08 30th International Conference on Software Engineering, May 10–18, Leipzig, Germany.

Zuk, T., Collins, C., & Carpendale, S. (2008). *How to effectively use interaction logs for usability evaluation purposes*. Paper presented at the BELIV '08 Proceedings of the 2008 conference on BEyond time and errors: novel evaluation methods for Information Visualization, April 5, Florence, Italy.

ABOUT THE AUTHORS

Hacı Ali Duru, holds a Bsc in Computer Engineering and an Msc on Industrial Engineering. He received his PhD from Turkish Military Academy in Operations Research. He currently works as a Software Development Team Manger. His primary research interests are multimodal human–computer interaction, information visualization and data mining.

Murat Perit Çakır, Ph.D., is an assistant professor in the Department of Cognitive Science of the Informatics Institute at Middle East Technical University. His main research areas involve computer-supported collaborative learning, human–computer interaction, interaction analysis, groupware design, math cognition, and cognitive neuroscience of learning.

Veysi İşler, Ph.D., is an associate professor in the Department of Computer Engineering at Middle East Technical University. His research interests cover a range of issues related to virtual environments (VEs), such as developing efficient algorithms for VEs, design and development issues in applications of VEs, and qualitative evaluation of VEs.