An Empirical Investigation on Documentation Usage Patterns in Maintenance Tasks

Gabriele Bavota[1], Gerardo Canfora[1], Massimiliano Di Penta[1], Rocco Oliveto[2], Sebastiano Panichella[1]
[1]University of Sannio, Benevento, Italy
[2]University of Molise, Pesche (IS), Italy
{gbavota, canfora, dipenta}@unisannio.it, rocco.oliveto@unimol.it, spanichella@unisannio.it

***Abstract*—When developers perform a software maintenance task, they need to identify artifacts—e.g., classes or more specifically methods—that need to be modified. To this aim, they can browse various kind of artifacts, for example use case descriptions, UML diagrams, or source code.**

This paper reports the results of a study—conducted with 33 participants— aimed at investigating (i) to what extent developers use different kinds of documentation when identifying artifacts to be changed, and (ii) whether they follow specific navigation patterns among different kinds of artifacts.

Results indicate that, although participants spent a conspicuous proportion of the available time by focusing on source code, they browse back and forth between source code and either static (class) or dynamic (sequence) diagrams. Less frequently, participants—especially more experienced ones—follow an "integrated" approach by using different kinds of artifacts.

I. Introduction

Maintenance tasks are generally facilitated when software documentation (e.g., the requirements specification, design document, test report, and user manual) is available [2], [10], [28]. Indeed, having documentation available during system maintenance reduces the time needed to understand how maintenance tasks can be performed by approximately 20% [28]. In addition, besides time reduction, documentation allows developers to find better and more accurate technical solutions to a given maintenance task [28].

Although several studies have shown the usefulness of documentation during maintenance tasks (see e.g., [2], [7], [10], [18], [26], [28]), it is still unclear how such documentation is browsed by developers to understand how the system should be modified to implement a specific change. At one extreme, one can argue for using all the available documentation, as each artifact is equally useful, since it provides a description of the system with different levels of details. Also, the documentation could be browsed starting from high-level artifacts (e.g., use cases) to low level artifacts (e.g., dynamic models). Even if there is an anecdotal evidence that such an approach could work, without a proper empirical investigation it remains only a conjecture. Also, different developers—with different skills and experience—might follow different paths. Thus, on one hand, guessing a priori navigational paths is quite challenging. On the other hand, understanding such paths is relevant not only to highlight the importance of high-level documentation, but also to help tool developers enhancing modelers and Integrated Development Environments (IDEs) to better support program comprehension activities by facilitating effective and efficient artifact navigation and browsing.

All these considerations motivate our work. We conduced a study, involving 33 participants—-among undergraduate and graduate students from different universities—aimed at analyzing to what extent developers use different kinds of documentation when identifying pieces of code (e.g., methods) to be changed and whether they follow specific navigation paths among different kinds of artifacts. In the context of our study, we asked participants to perform 8 different maintenance tasks on a Java software system. Besides source code, participants had available use case descriptions, sequence diagrams, class diagrams, and Javadoc. We used an Eclipse plugin to capture how much time was spent by participants on different artifacts, and how they navigated from an artifact to another.

The obtained results indicated that—even if a substantial proportion of time (about 80% on average) is spent on source code, participants also browsed back and forth between source code and either static (class) or dynamic (sequence) diagrams, the latter being more used than the former. Less frequently, participants—and in particular those with a higher degree of experience, i.e., graduate students—follow an "integrated" approach, in which different kinds of artifacts were used, for example starting the task from use cases, then browsing sequence and/or class diagrams before accessing the source code. Such results could be used to enhance IDEs with a recommendation system able to suggest a particular navigation path aiming at facilitating the browsing of the available documentation. Such a recommender might be particular useful in large systems where the browsing of myriad software artifacts could represent an obstacle instead of a facilitation when performing the maintenance task [8], [15].

Paper organization. Section II presents the definition and planning of our study, while Section III discusses the results achieved. Section IV presents the threats that could affect the validity of our study. Finally, after a discussion of the related literature (Section V), Section VI concludes the paper outlining direction for future work.

II. Study Definition and Planning

This section describes the design and planning of the our empirical study. The *goal* of the study is to observe how developers browse different kinds of software artifacts, with the *purpose* of understanding how they build knowledge needed to deal with a maintenance task and, specifically, to identify classes and class elements (methods and attributes) that need to be changed when performing a maintenance task. The *perspective* is of researchers interested to identify relevant navigation paths across artifacts that result helpful

during a software evolution task. This result can be used, for example, to build smart recommenders that guide developers by suggesting navigations across artifacts or to better organize and index the documentation available for a software project.

A. Context Selection

The study involved 33 participants, selected entirely on a voluntary basis—i.e., using a convenience sampling—mainly among undergraduate students of the Computer Science Degree at the University of Molise, and among master students, PhD students (including visiting students) of the Computer Science Engineering Degree of the University of Sannio. Overall, 11 Bachelor students, 18 Master students, and 4 PhD students participated to the study. Master and PhD students had already experience on some industrial or research projects, as well as on the development and maintenance of complex software systems.

The objects which the tasks were performed on are use case descriptions, design level sequence and class diagrams, Javadoc, and Java source code files of a school automation system, named SMOS, developed by graduate students at the University of Salerno (Italy). SMOS offers a set of features aimed at simplifying the communication between the school and the student's parents. The system is composed of 121 classes with their respective Javadoc for a total size of 23 KLOC. The documentation is represented by 67 use cases, 72 design level sequence diagrams, and 6 design level class diagrams. Each class diagram represents the relationships between all the classes involved in a specific subsystem, e.g., teaching management.

In the context of our study, we asked participants to perform 8 different maintenance tasks on SMOS, of which 3 were bug-fixing tasks, 3 related to add a new feature, and 2 related to improve existing features, i.e., performing a perfective maintenance task.

B. Research Questions

The study aims at investigating the following research questions:

- **RQ$_1$:** *How much time did participants spend on different kinds of artifacts?* This research question aims at analyzing the time spent by participants on the different kinds of available artifacts. On the one hand, artifacts used for less time can be thought of being less useful. On the other hand, some artifacts intrinsically require more time to be read (e.g., source code) while for others (e.g., use cases, sequence diagrams) a quick look may just suffice to provide a useful piece of information.
- **RQ$_2$:** *How do participants navigate different kinds of artifacts to identify code to be changed during the evolution task?* This research question is the core of our study aimed at analyzing the sequences of interactions made with different artifacts. In particular, we will investigate (i) how do participants start the task, (ii) what kinds of artifacts do they browse before getting to the source code, and (iii) whether there are frequent browsing patterns, e.g., repeated navigation back and forth between source code and class diagrams.

TASK DESCRIPTION

In SMOS a registered user can have six different roles: Admin, Teacher, Student, Parent, Janitors, and Director. Suppose that we want to remove the "Director" role, which changes do you need to made on source code? Specify for each involved class/method the changes you would apply.

QUESTIONS

Write the list of methods modified to perform this task specifying how you modified these methods. For example: "application.userManagement.UpdateUser.doGet". I added the line of code "x=3;" after the line of code "y++;".

Write the list of attributes modified to perform this task.
For example: "bean.User.UID".

Fig. 1. Example of task description and related questions.

For each research question, we also analyzed the impact of participants' experience on the use and the navigation of the software documentation.

C. Study Procedure and Material

Before the study, we explained to participants what we expected them to do during their tasks. Specifically, we asked them to identify methods and attributes to be changed when performing each change task. We provided an overview of what kinds of artifacts they have available, briefly summarizing the purpose of each of them.

After illustrating the study, we gave participants up to 3 hours of time to perform the task. Note that it was not our intention to measure the task efficiency, hence we were not strict with the time. We only made sure participants properly performed the task, without collaborating.

We provided each participants with a customized Eclipse installation containing:

- The Java Development Environment (JDT) with the SMOS software system already imported together with its documentation, i.e., sequence diagrams, class diagrams, use cases, and Javadoc.
- FLUORITE[1] (Full of Low-level User Operations Recorded In The Editor), an Eclipse plug-in able to capture all of the low-level events when using the Eclipse editors. FLUORITE keeps track of all of the events that occur in the Eclipse editors also storing timestamps for each event. All data is saved in an XML log file.
- The Pdf4Eclipse[2] plug-in (used to visualize use cases, sequence diagrams, and class diagrams).
- An Eclipse HTML Editor[3] plug-in, used to visualize the Javadoc files.

Also, we provided participants with an URL of a page on ESurveysPro[4], a online survey tool we used to collect participants' answers.

[1] http://www.cs.cmu.edu/ fluorite/
[2] http://borisvl.github.io/Pdf4Eclipse/
[3] http://amateras.sourceforge.jp/cgi-bin/fswiki_en/wiki.cgi?page=EclipseHTMLEditor
[4] http://www.esurveyspro.com/

During the study, we instructed participants to access the ESurveysPro page and, for each of the eight tasks to be performed, to work following this procedure:

1) Access the page describing the task, and read the task description.
2) Then, use Eclipse to find a solution for the task (without however applying the change).
3) Answer the questions in the opened ESurveysPro page. For each task, participants had to provide, using two different form fields, the list of methods and instance variables (attributes) that need to be modified. Fig. 1 shows an example of task description and questions being asked for the task. For each question examples of answers are provided. We made clear to participants that example answers are not related to the task, thus they are not valid answers.

After having completed the 8 tasks, participants had to fill a post-study questionnaire. The post-study questionnaire asked participants an opinion about the usefulness of the various kinds of artifacts, using a Likert scale [17] ranging between 1 (totally useless) and 5 (very useful). We also asked participants to provide a comment for the rank assigned to each kind of artifact.

D. Data Collection

After tasks were completed, we collected from each participant (i) the XML logs generated by FLUORITE; and (ii) the answers provided on ESurveysPro. Concerning FLUORITE logs, they have been parsed through a Java tool developed on purpose. The tool extracts, for each task performed by each participant, the ranked list of documents explored during such a task together with the time spent on each document. An example of generated list is:

$$UseCase(27) \rightarrow SequenceDiagram(48) \rightarrow Code(82)$$

indicating that the participant started by reading an use case description for 27 seconds, moving then to a sequence diagram for 48 seconds, and finally access the source code for 82 seconds.

We pruned out from such logs browsing activities shorter than 5 seconds. Such a threshold was set by observing how subjects navigated source code during the tasks. Although this would remove some potentially useful information, we assume that such short activities are mainly due to the need for scrolling across various windows in the IDE.

E. Analysis Method

To answer RQ_1, we measure (in seconds) the time spent by participants on each of the artifact types considered in our study (i.e., the four different documentations plus source code). We also analyze the scores provided by the participants in the post-survey questionnaire to indicate their perceived usefulness of the exploited artifacts. Results are reported in terms of descriptive statistics and boxplots.

Besides analyzing the whole dataset collected during our study, we investigate whether participants with different levels of experience (graduate *vs.* undergraduate students) use artifacts differently. Due to the limited number of PhD students, and also for the sake of simplicity, we just distinguish between undergraduate (i.e., bachelor) and graduate (i.e., Master or PhD) students. The main reason why we analyzed results of graduate and undergraduate students separately is because the former had (i) some real working experience, and (ii) in all cases, some experience in the development and maintenance of complex projects, which would often favor the need for using high-level documentation when performing a comprehension task.

In addition to descriptive statistics and boxplots, we use Mann-Whitney test [6] to compare the proportion of time spent on each kind of diagram by participants having different levels of experience. We use a non-parametric test because the Shapiro-Wilk normality test indicated that data—related to all kinds of artifacts for both undergraduate and graduate—deviate from a normal distribution (p-value < 0.001 in all cases). We also evaluate the magnitude of the observed differences using the Cliff's Delta (or d), a non-parametric effect size measure [12] for ordinal data. We followed the guidelines in [12] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Still in the context of RQ_1, we verify if there is a correlation between the kind of artifacts exploited by participants and the correctness of the performed tasks. Note that our study does not aim at investigating whether the usage of different artifacts influences the task correctness. This cannot be done, because it would have required a specific controlled experiment with participants receiving different treatments, e.g., using some diagrams only, or "forced" to follow specific navigational paths only. Instead, this analysis should be considered as a form of sanity-check, to determine whether participants performed tasks seriously and whether participants using more specific kinds of artifacts could have suffered particular problems.

To measure the completeness and correctness of the tasks performed by each participant (i.e., her ability in correctly individuating the code components impacted by a maintenance activity), we used a combination of two well-known Information Retrieval metrics, recall and precision [3]. Recall measures the percentage of code components actually impacted by a maintenance activity correctly identified by a participant, while precision measures the percentage of identified components that are actually impacted. Since recall and precision measure two different (but related) concepts, we use their harmonic mean (i.e., F-measure [3]) to obtain a balance between them when measuring task correctness.

The correlation between the type of artifacts exploited by participants and the correctness and completeness of the performed tasks is computed through (i) the Spearman correlation, performed between the time spent by participants in each task on each type of artifact and the correctness achieved in the task, and (ii) by building a logistic regression model for correctness based on the use (or not) of different kinds of artifacts.

Concerning RQ_2 we extracted, using the data derived by the FLUORITE plugin, information concerning how participants navigate different artifacts, and specifically:

- What artifacts did participants looked first, i.e., where the comprehension task started. Usually, one assumes this

TABLE I. Recall, Precision, and F-measure achieved by participants when performing the tasks.

Dataset		Recall	Precision	F-measure
Undergraduates	Mean	0.65	0.79	0.71
	Median	0.81	1.00	0.82
	St. Dev.	0.40	0.38	0.37
Graduates	Mean	0.67	0.88	0.76
	Median	0.88	1.00	0.93
	St. Dev.	0.37	0.31	0.35
All	Mean	0.67	0.85	0.75
	Median	0.88	1.00	0.86
	St. Dev.	0.38	0.34	0.36

starts from requirements/use cases, although there are developers that start from source code directly.

- What artifacts did participants browse before getting to source code. This could potentially indicate the pattern followed to locate the source code element to be changed.
- What is the likelihood of making a transition from one kind of artifact to the other. This can likely indicate how the information gained by browsing a certain kind of artifact raises the need for accessing another kind of artifact, e.g., browsing source code after accessing sequence or class diagrams, or else looking at static models after dynamic models.
- What are the most frequently followed patterns. This was done by matching regular expressions of length varying from two to four onto the mined logs, and determining for each pattern whether it was iterated, e.g., participants could go back and forth between source code and class or sequence diagrams repeatedly.

Finally, we investigated whether participants with different levels of experience followed different patterns and whether following certain patterns can influence the task correctness.

All statistical analyses of this paper have been performed using the *R* environment [19]. For all statistical procedures, we assumed a significance level of 95%.

F. Replication package

To facilitate the replication of this study, a complete replication package is available[5]. It includes (i) an Eclipse installation bundle, with all the exploited plug-ins installed and the object system SMOS (source code and other artifacts) already imported, (ii) the task description for all 8 tasks, (iii) the post-study questionnaire, and (iv) the FLUORITE logs for the 33 participants. Also, the package includes the working data set with our study results.

III. Analysis of the Results

Before answering the research questions formulated in Section II-B, it is important to verify whether participants seriously performed the assigned tasks. To this aim, Table I reports the average values for recall, precision, and F-measure achieved by undergraduate and graduate students, as well as when considering the entire dataset. Results show that participants were able to achieve quite good performances, with an average F-measure of 0.75. This sanity check makes us confident that participants seriously performed the assigned tasks. Also, as expected, graduate students achieved, on average, better performances than undergraduate students (+5% in terms of F-measure).

[5]http://distat.unimol.it/reports/icsm-docs/

TABLE II. Use (percentage of tasks and time spent) of different kinds of artifacts: descriptive statistics.

Artifacts	Tasks (%) (All)	Tasks (%) Undergrad.	Tasks (%) Graduate	Time spent (all data, %) mean	1Q	median	3Q
Use case	33	28	36	3	0	0	2
Sequence Diagram	72	68	74	10	0	7	16
Class Diagram	60	49	66	13	0	4	15
Javadoc	15	21	11	2	0	0	0
Source Code	100	100	100	72	66	79	89

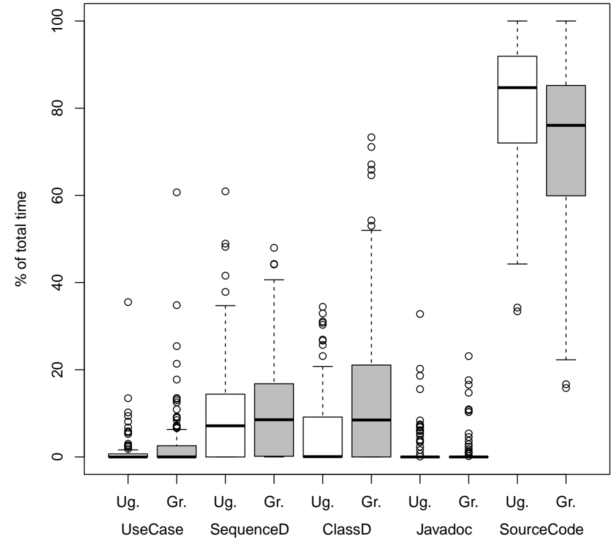

Fig. 2. Usage (in percentage) of different kinds of artifacts. Ug = undergraduate students, Gr = graduate students.

A. RQ1: How much time did participants spend on different kinds of artifacts?

Table II reports the percentage of tasks in which each kind or artifact has been used (for the entire dataset as well as by separately considering participants with different levels of experience), and descriptive statistics about the percentage of time spent on various kinds of artifacts (by considering the entire dataset). Fig. 2 shows boxplots of such percentage for different levels of experience.

If considering the whole dataset, and analyze the time spent on artifacts (right-side of Table II), results indicate that participants spent most of their time (72% on average) on source code. Our conjecture—partially supported by what we observed during the tasks and ny talking with participants—is that this might be due to two reasons. First, even when participants were able to identify the impacted components by analyzing documentation artifacts we observed that they checked-back in the source code that the identified methods/attributes were actually there and really impacted by the maintenance activity to perform. This suggests a kind of distrust with respect to documentation artifacts, as also confirmed by the fact that source code has been used in 100% of the tasks. Second, source code clearly requires more time to be read and understood as compared to the artifacts present in the documentation. In particular, participants spent, on average, 154 seconds on each

TABLE III. PERCENTAGE OF TIME SPENT ON ARTIFACTS BY PARTICIPANTS WITH DIFFERENT EXPERIENCE: MANN-WHITNEY TEST AND CLIFF'S d EFFECT SIZE (POSITIVE VALUES INDICATE DIFFERENCES IN FAVOR OF GRADUATE STUDENTS, NEGATIVE IN FAVOR OF UNDERGRADUATES).

Artifact	p-value	Cliff's d
Use Case	0.1020	0.1030
Sequence Diagram	0.3102	0.0749
Class Diagram	**0.0001**	0.2757
Javadoc	0.0268	-0.1040
Source Code	$<$ **0.0001**	-0.2939

source code file, compared to the 70 spent on a class diagram, 49 on a Javadoc file, 35 on a sequence diagram, and 34 on a use case.

If we look at the percentage of tasks in which each kind of artifact was used at least once (left-side of Table II), we notice that—besides source code, obviously used in 100% of the tasks—the most commonly used documentation artifacts are class and sequence diagrams. The latter were used in 72% of the task. On such diagrams, participants spent on average 10% of their time (median=7%). Only one of the 33 participants did not exploit at all sequence diagrams during the tasks and justified such a choice in the post-study questionnaire: *"sequence diagrams would be useful only if class diagrams were not present"*. However, as we will see shortly, this is an isolate point-of-view.

As for class diagrams, they were used in 60% of tasks and participants spent, on average, 15% of their time on them (median=4%). This strong misalignment between the mean and the median values for class diagrams highlights that, while generally they are used for a lower proportion of time as compared with sequence diagrams, some participants spent a very high proportion of their time on class diagrams, as also shown by the outliers reported in Fig. 2. Two participants did not use at all class diagrams in the tasks.

Turning to use cases, they were used in 33% of tasks by participants, which focused on them just the 3% of their time, on average. As said before, participants spent just 34 seconds, on average, on each consulted use case against, for instance, the 154 spent on each source code file. Among the 33 participants, three of them did not access at all use cases.

Finally, Javadoc documentation was not used a lot by participants of our study. They accessed Javadoc in just 15% of the tasks. Also, 11 participants out of 33 never open Javadoc files during the tasks.

Concerning the time spent by participants with different experience levels on different artifacts, Fig. 2 and the results of the Mann-Whitney test reported in Table III indicate that: (i) there is no significant difference in accessing use cases and sequence diagrams; (ii) graduate students use class diagrams significantly more than undergraduates, with a medium effect size; (ii) undergraduates students used source code and Javadoc significantly more than graduate students, with a small and medium effect size respectively. Such results partially contradict those of other studies [20], which indicated that junior developers tend to benefit of models than senior developers, that tend to directly focus onto source code.

To better understand the results of the quantitative analysis,

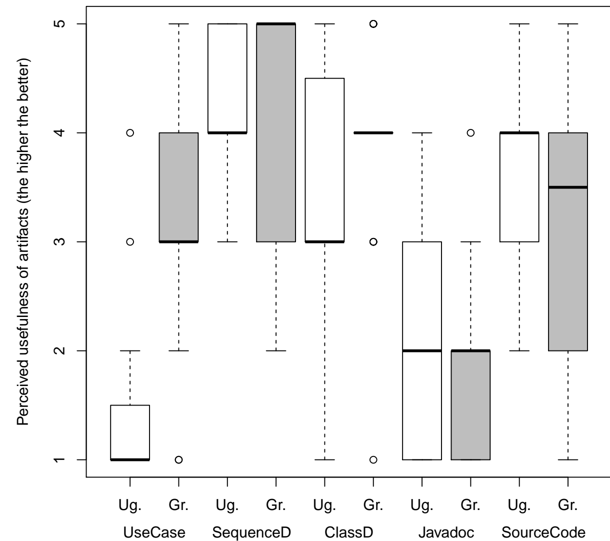

Fig. 3. Perceived usefulness of the different kinds of artifacts as indicated by participants. Ug = undergraduate students, Gr = graduate students.

we analyzed the feedbacks provided us by means of the post-study questionnaire. Fig. 3 shows boxplots—for different levels of experience—of the ratings provided by participants to the usefulness of the different kinds of artifacts used. As explained in Section II-C, one corresponds to classify a kind of artifact (documentation as well as source code) as "totally useless", while five indicates a "very useful" kind of artifact.

As we can notice, sequence diagrams are considered to be the most useful kinds of artifact, with a mean score of 4.3 for both undergraduates and graduates (median 4 for undergraduates and 5 for graduates). Some of the comments left by participants in the post-study questionnaire explain the reasons behind this evaluation. Several of them explained how *"once found the sequence diagram(s) describing the feature(s) involved in a change request, it was easy to identify the candidate impacted components. This strongly speeds up the tasks."* Others explained as sequence diagrams *"represent a fair compromise between use cases (too abstract) and class diagrams (providing useless details about an entire subsystem)"*.

Class diagrams and source code were generally ranked as equally useful. However, while undergraduates found source code slightly more useful (mean 3.6, median 4) than graduates (mean 3.2, median 3.5), the opposite happens for class diagrams, that were found more useful by graduate students (mean 3.9, median 4) than by undergraduates (mean 3.4, median 3). Among the 8 participants that considered class diagrams very useful, five of them explained as *"it is easy to map class diagrams on source code, and thus to fast check the candidate impacted components identified from the diagram."* Five of the 33 participants declared the source code as the most useful artifact. The perceived reason is that: *"while the provided high-level documentation is useful to speed-up the task, consulting source code is mandatory to perform some of them, like the required bug-fixes."*

As for use cases, it is interesting to note that the usefulness

assessment provided by graduates (mean 3.2, median 3) is higher than for bachelor (mean 1.5, median 1). This is the only case for which the Mann-Whitney test reveals a statistically significant difference (p-value=0.002, Cliff's d 0.68 – high), while for all other artifacts the differences between the two levels of experience are not significant. This suggests how more experienced participants are able to start the task from requirements/use cases before accessing models and source code. Undergraduates failed to explain use cases, as they tried to identify object names within them *"in use cases it was not possible to find information about components of the system impacted by a change"*, rather than relying on use cases to identify the piece of functionality to be changed before accessing sequence/class diagrams.

Finally, the provided feedbacks confirmed that Javadocs were perceived as the least useful artifacts. For Javadoc, the mean and median score was 2 ("useless") for both undergraduate and graduate students. Participants declared that *"with the other sources of documentation available Javadoc became useless to identify impacted components."* This is to say, our study does not show that Javadoc is useless: it is likely to be very useful during development activities, e.g., when using a new API. Instead, it provides a limited (or no) support when analyzing the impact of a change.

As explained in Section II-E, we also analyzed the presence of possible correlations between the time spent by participants on the different kinds of artifacts and the correctness of the performed tasks in terms of recall, precision, and F-measure. By applying the Spearman correlation test no interesting correlations were found for undergraduate and graduate students, as well as when considering all participants as a single dataset. Also, a logistic regression model for correctness based on the use (or not) of different kinds of artifacts did not lead to any significant result, i.e., none of the artifacts resulted significant in the model.

Summary for RQ$_1$.

1) Participants spent more time to analyze low-level artifacts as compared to high-level artifacts.
2) Participants consider sequence diagrams as the most useful source of documentation when performing the required tasks, followed by class diagrams and source code.
3) Undergraduate students spent a significantly higher proportion of time on source code than graduate students who, instead, spent more time on class diagrams.

B. RQ2: How do participants navigate different kinds of artifacts to identify code to be changed during the evolution task?

Table IV reports, for each kind of artifact used in our study, the number and percentage of tasks participants started from such artifact. The most frequent starting point is by far source code (42% of the tasks), followed by sequence diagrams (25%), class diagrams (17%), use cases (12%), and Javadoc (3%). Note that this result is quite surprising since one could expect that developers start their analysis from high-level artifacts going down to the code. Instead, in our study 84% of the tasks started from source code and design models, i.e., class or sequence diagrams.

TABLE IV. WHAT PARTICIPANTS LOOKED FIRST.

Artifact	All data		Undergrad.		Graduates	
	#of Tasks	Perc (%)	# of Tasks	(%)	# of Tasks	(%)
Use Case	31	11.92	3	3.16	28	16.97
Sequence Diagram	66	25.38	23	24.21	43	26.06
Class Diagram	45	17.31	9	9.47	36	21.81
Javadoc	9	3.46	4	4.21	5	3.03
Source Code	109	41.92	56	58.95	53	32.12

When observing data for different levels of experience (right-side of the table), what we notice is pretty consistent with findings of **RQ**$_1$ concerning the proportion of usage for different kinds of diagrams. Basically, undergraduates tend to start tasks mainly using source code (58%), while this percentage is only 32% for graduates. The percentage of participants starting with sequence diagrams is similar (24% for undergraduates, 26% for graduates), while graduates tend to start with class diagrams more than undergraduates (22% vs 9%). Finally, there is a non-negligible proportion of graduates that starts from use-cases (17%, vs. 3% of undergraduates). This is likely due to the fact that graduate students have a better training on software engineering principles and on how using models and high-level artifacts during maintenance tasks, and also because they have more experience in evolving existing systems.

TABLE V. PATTERNS FOLLOWED BEFORE REACHING SOURCE CODE.

Pattern	All data		Undergrad.		Graduates	
	# of Tasks	(%)	# of Tasks	(%)	# of Tasks	(%)
S	36	13.85	17	17.89	19	11.51
D	35	13.46	8	8.42	27	16.36
(SD)+	22	8.46	2	2.10	20	12.12
(US)+	18	6.92	2	2.10	16	9.70
U(SD)+	7	3.46	1	1.05	6	3.64
(DS)+	7	2.69	1	1.05	6	3.64
J	4	1.54	3	3.16	1	0.60
U	4	1.54	0	0.00	4	2.42
S(US)+	3	1.15	1	1.05	2	1.21
SU(SD)+	2	0.77	0	0.00	2	1.21
Other	13	5.00	4	4.21	9	5.45

S = Sequence Diagram, D = Class Diagram
U = Use Case, J = Javadoc

Since we found that in 58% of cases source code does not represent the entry point, we analyzed what are, in these cases, the pattern followed by participants before reaching source code. Table V reports all possible patterns followed by participants (using through regular expressions). In a similar proportion of tasks, participants access sequence or class diagrams before going to source code. This happens in 71 tasks, 36 for sequence (14%) and 35 for class (13%) diagrams.

Another frequently followed path consists of one or more switches between sequence and class diagrams. This path is more frequent starting from the sequence (22 tasks)—row (SD)+ in Table V—than from class diagrams (7 tasks)—row (DS)+ in Table V. In both cases, participants tried to gain source code knowledge from its most direct model representations (i.e., class and sequence diagrams) before going through it. Also, for 18 tasks, participants switch one or more times between use case (used as starting point) and sequence diagrams—row (US)+ in Table V. Overall, it is interesting to

TABLE VI. AVERAGE TRANSITION FREQUENCIES BETWEEN THE KINDS OF ARTIFACTS.

From/To	U	S	D	J	C
U		56%	8%	0%	36%
S	5%		17%	1%	77%
D	2%	18%		2%	78%
J	0%	6%	16%		78%
C	7%	49%	37%	7%	

S = Sequence Diagram, D = Class Diagram
U = Use Case, J = Javadoc, C = Source Code

note that sequence diagrams are accessed in four out of the five most frequent path followed before reaching source code. Other paths reported in Table V are quite uncommon, e.g., opening a use case (row U) or a Javadoc file (row J).

When looking at results by different levels of experience (right-side of Table V), it can be noticed that, besides what it is known already from previous analyses, graduate students use much more navigation patterns across different kinds of diagrams. As the table shows, undergraduates just looked at sequence or class diagrams before diving into source code. Instead, graduate students also followed more complex navigation patterns, e.g., sequence+class (with some iterations), use case+sequence (with some iteration), or even use cases followed by iterations on sequence and class diagrams. Once again, this indicated that people with more experience are more prone to follow an "integrated" approach when performing a comprehension task.

Then, we analyzed the transition frequencies between the different kinds of artifacts used in our study. Table VI reports the results considering the entire dataset. As it can be noticed, the most frequent transitions are toward the source code (column C), 77% of which are from a sequence diagram, and 78% from class diagrams and Javadoc files. The take-away of these results is that, after have gathered information from one of those kinds of artifacts, developers try to map them into source code elements. Note that this is true also when separately analyzing participants having different experience levels with small changes in the transition frequencies.

The behavior of participants when reading use cases is, instead, pretty different from the one observed above. They shift toward source code in just 36% of times, privileging the reading of a design diagram (64% of the cases, 56% for sequence and 8% for class diagrams) before reaching source code. However, when analyzing the data for participants having different experience, some differences came out. In particular, graduate students tend to consult a low-level diagrams after accessing an use case (72%, 64% for sequence and 8% for class diagrams), against the 43% of undergraduates (35% for sequence and 8% for class diagrams). After reading an use case, undergraduates go to source code in 56% of cases, against the 27% of graduates. This further confirms that more experienced developers are more prone to use different sources of documentation when performing a comprehension task.

Other common transitions between different kinds of artifacts occur (i) when reading a sequence diagram toward a class diagram (17%) and (ii) when reading a class diagram toward a sequence diagram (18%). In this case, no interesting difference has been observed between participants having different experience.

TABLE VII. MOST FREQUENT NAVIGATIONAL PATTERNS.

Pattern	All data		Undergrad.		Graduates	
	Occ.	(%)	Occ.	(%)	Occ.	(%)
USDC	12	4.62	0	0.00	12	7.27
USD	21	8.08	2	2.11	19	11.52
USC	35	13.46	10	10.53	25	15.15
UDC	13	5.00	3	3.16	10	6.06
SDC	55	21.15	15	15.79	40	24.24
SC	153	58.85	58	61.05	95	57.58
DC	128	49.23	39	41.05	89	53.94

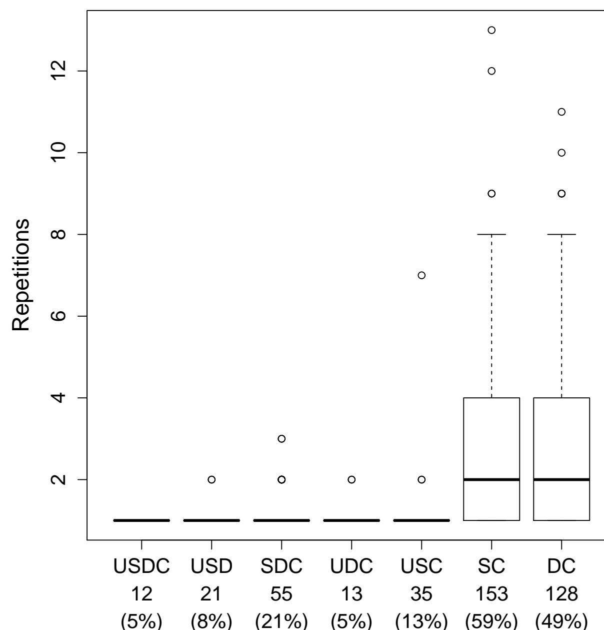

Fig. 4. Most frequent navigational patterns and distribution of their repetitions. S = Sequence Diagram, D = Class Diagram, U = Use Case, C = Source Code.

It is also interesting to analyze what other artifacts participants access immediately after browsing source code. Table VI indicates that participants go back from source code to documentation just to access design diagram, i.e., sequence (49%) and class (37%) diagrams. Again, no important differences were found between participants having different experience.

Finally, we analyzed the most frequent navigational patterns followed by participants during the tasks. Table VII and Fig. 4 report information about the seven most frequent patterns we found. In particular, Table VII reports the number and percentage of occurrences on the whole dataset and for participants with a different degree of experience, whereas Fig. 4 shows the boxplots for the distribution of its repetitions (i.e., the number of times a pattern appears in a single task). As it can be expected according to what observed so far, the most frequent pattern consists of going back and forth from sequence diagram to source code: this occurred in 153 tasks (59%). The median of its repetitions is two, but we also found cases where this pattern has been repeated more than 10 times in a single task. Another very frequent pattern is that going back and forth from class diagrams to source code, present in 128 tasks (49%) with also a median repetition of two. Among the longer patterns (i.e., those having a length > 2), the most

frequent is that going from sequence to class diagram and then to source code (SDC in Fig. 4). This pattern has been followed by participants in 21% of the performed tasks, generally with a single repetition. Also, in 13% of tasks, participants went from use cases toward sequence diagrams, and finally to the code. In addition, from the analysis of Fig. 4 we can conclude that (i) Javadoc is not present in any of the most common patterns; and (ii) all common patterns end (as expected) with a source code artifact.

When looking at the occurrences of patterns among participants with a different level of experience (right-side of Table VII), we can notice that (i) the SC pattern (sequence+code) is consistently followed by about 60% of both undergraduates and graduates; (ii) patterns involving use cases (USDC, USD, USC, and UDC) are much more frequent for graduate than for undergraduates; and (iii) for what concerns longer patterns followed by undergraduates, the SDC pattern was followed in 16% of the cases, and USC in 10% of the cases. In summary, we can notice a higher proportion of patterns reflecting a more "integrated" approach for graduates. Also, graduate students followed patterns involving class diagrams and code (DC) more (54%) than undergraduates (41%). We did not notice any significant difference in the number of iterations for all the above mentioned patterns, except for the SC pattern, that received a median of 3 iterations for undergraduates, that used it and of 2 iterations by graduates that used it. The difference is statistically significant (p-value =00017) and the Cliff's d effect size medium ($d = 0.293$). In other words, less-experienced participants had to go back and forth between sequence diagrams and source code more than experienced ones to locate the methods to be changed.

As done for RQ_1, we also statistically verified the relationship between the patterns followed by participants and the correctness of their tasks. In particular, we built a logistic regression model for correctness with respect to the use (or not) of the different patterns. Also in this case, we did not find any statistically significant result.

Summary for RQ_2.

1) Participants tend to start the assigned task from source code or from design documents, i.e., class and sequence diagrams.
2) More experienced participants tend to follow a more integrated approach than less experienced ones, traversing different kinds of diagrams, e.g., starting from use cases, and then browsing design documents, until reaching source code.
3) During their task, participants tend to go back and forth repeatedly between source code and to design diagrams (sequence and class diagrams).

IV. THREATS TO VALIDITY

This section discusses the threats that could affect our results.

Threats to *construct validity* concern the relation between the theory and the observation. In our study, this threat can mainly be due to errors in the collected measurements. For what concerns capturing participant's browsing activities, we relied on an existing tool (FLUORITE), making sure each participant had correctly installed it, and carefully instructed them how to browse artifacts in Eclipse while using the tool. When collecting results, we discarded cases of short access to artifacts (less than five seconds) that are unlikely to be an indication of reading the document, but rather of scrolling different documents. Clearly, this might have meant loosing some quick, but valid, accesses.

Another threat concerns the way the correctness of the task is evaluated, i.e., by means of precision, recall, and F-measure computed over the list of elements to be modified as identified by participants. On the one hand this allows a subjective evaluation and allows to perform a comprehension task without requiring the execution of source code. On the other hand, this can provide a coarse-grained and partial evaluation of how the comprehension task was performed.

Threats to *internal validity* concern any confounding factor that could influence our results. For example, such a threat may be due to the fact that some participants might have decided not to browse diagrams because they were unreadable or the tool was not usable. To mitigate such a threat, we avoided to use any specific UML modeler (we used PDF documents instead), and we produced diagrams large enough to be easily readable.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. As explained in Section II, this is more an observational study rather than a controlled experiment, as all participants received the same treatment. Wherever possible, however, we used appropriate statistical procedures and effect size measure to support our claims.

Threats to *external validity* concern the generalization of our findings. This study has been conducted with students, and for this reason the obtained results may not generalize to professionals, which might be used to perform comprehension task using high-level artifacts in a different way, or in some cases not using them at all. To some extent, our participants can be considered as representative of junior developers, joining a project as newcomers to perform a maintenance task.

Another threat to external validity is related to the use of source code and documentation related to a single project. We do not know whether the comprehension of other kinds of projects would benefit of navigation patterns different than those discovered in this paper.

V. RELATED WORK

Several studies have been performed to analyze the benefits of UML documentation during software development and evolution [5]. In the next section we focus the attention on studies analyzing the effect of documentation on maintenance/comprehension tasks. In addition, we also discuss study carried out to analyze the behavior—from different perspectives—of developers performing maintenance tasks.

A. Impact of UML documentation on Maintenance Tasks

Experiments aimed at studying the impact of UML documentation in software maintenance [2] indicated that such a

documentation improves the functional correctness of changes and the quality of the design. While simple class diagrams, with or without stereotypes, help low ability or low experience participants, a complete, thorough UML documentation requires a certain learning curve to become useful [2]. In fact, in some cases the previous experience of participants influences the understandability of UML diagrams. Torchiano [27] showed that object diagrams have a significant impact on comprehension tasks, when compared with UML documentation consisting of class diagrams only.

Dzidek *et al.* [10] performed a controlled experiment aimed at investigating the costs of maintaining and the benefits of using UML documentation during the maintenance and evolution of software systems. In the context of the experiment, participants (represented by professional developers) performed evolutionary tasks with and without UML documentation. Their results indicated that participants using UML documentation were able to statistically increase the correctness of changes. Also, they were able to slightly improve the design quality at the expense of an insignificant increase in development time caused by the overhead of updating the UML documentation.

UML limitations in aiding program understanding are highlighted in experiments performed by Tilley and Huang [26]. They highlighted that UML does not provide a sufficient support to represent domain knowledge. Lemus *et al.* [7] showed that composite states improve the understandability of statecharts provided that participants had a previous experience in using them. This also happens when UML is complemented with complex formalisms, such as the Object Constraint Language (OCL) [4]: a substantial training is required to make OCL useful, although for some tasks OCL better helped low ability participants, who were not able to guess system functionality from the textual description.

The role of dynamic UML diagrams in software comprehension was investigated by Otero and Dolado [18]. The comprehension level and the time required to perform the comprehension task resulted different for different diagrams and system complexities. Abrahão *et al.* [1] also analyzed the support given by sequence diagrams during the comprehension of functional requirements. The results showed that sequence diagrams improve the comprehension of the functional requirements in the case of high ability and more experienced participants.

We share with the aforementioned studies the need to analyze the support given by software documentation during software evolution. However, we did not focus on a specific kind of documentation. Instead, we provided to participants several documentation artifacts aim at studying which are the most used artifacts and how developers use such artifacts.

Tryggeseth [28] conducted a study, for some aspects similar to our, to analyze the impact of the availability of up-to-date documentation on maintenance tasks. In the context of the experiment participants were asked to perform maintenance tasks with and without software documentation (represented by requirements specification, design document, test report, and user manual). Their results indicated that participants using the available documentation spent less time to understand how to implement a change request. Besides reducing the time, the documentation also allowed participants to better understand the system and provided more detailed solution on how to incorporate the changes.

While we share with Tryggeseth the need for analyzing the impact of several software documentation artifacts on maintenance tasks, our study presents two main differences: (i) we analyzed how developers use documentation during software evolution aimed at identifying particular navigation paths; and (ii) we also investigated the effect of experience on how participants follow different usage paths.

B. Studies about Developers' Behavior during Maintenance Tasks

von Mayrhauser and Vans [29] observed how professional developers work when performing maintenance tasks, finding that programmers use a multi-level approach during source code understanding, switching between different programs as well as between different sources of documentation.

Robillard *et al.* [21] performed an exploratory study to analyze the factors that contribute to effective program investigation behavior, while Sillito *et al.* [23] performed two qualitative studies aimed at understanding what a programmer needs to know about a code base when performing a change task, how a programmer goes about finding that information, and how well today's programming tools help in that process.

Singer *et al.* [24] studied the daily activities of developers. Such a study provides some guidelines fro tool designers that represent an alternative to the traditional paths taken in human-computer interaction, namely those issuing from the study of the users' cognitive processes and mental models, and the emphasis on usability. Also, DeLine *et al.* [9] identified several usability issues of conventional development environments when a developer has to update a software system, including maintaining the number and layout of open text documents and relying heavily on textual search for navigation.

de Alwis and Murphy [8] analyzed how programmers experience disorientation when using Eclipse, identifying three factors that may lead to disorientation: the absence of connecting navigation context during program exploration, thrashing between displays to view necessary pieces of code, and the pursuit of sometimes unrelated subtasks.

Storey *et al.* [25] performed a study aimed at analyzing whether program understanding tools enhance or change the way that programmers understand programs. Based on the results achieved the authors suggested that tools should support multiple strategies (top-down and bottom-up, for example) and should aim to reduce cognitive overhead during program exploration.

The behavior of software developers has also been analyzed aimed at identifying approaches able to reduce the information overload (e.g., number of artifacts to be analyzed) of developers by filtering and ranking the information presented by the development environment [11], [15], [16]. The findings of our paper can complement such models. The usage patterns identified in our study can be used to complement such approaches providing a more effective support during program comprehension.

Recently, eye tracking systems have been used to investigate the comprehension of UML diagrams [13], [30], the

effect of the layout on the comprehensibility of software documentation artifacts [22], and the effect of design patterns on comprehension [14]. The use of eye tracking systems is particular useful to investigate on the way developers look at the documentation aimed at deriving guidelines for facilitating the comprehension of software documentation.

We share with all these studies the need to empirically analyze the behavior of developers during software development and maintenance. However, we analyzed the behavior from a different perspective. Specifically, our analysis aimed at analyzing how developers use software documentation in order to identify recurring usage paths. Such paths could be used to enhance contemporary IDEs and provide more effective strategies for browsing documentation artifacts.

VI. Conclusion and Future Work

This paper reported a study aiming at investigating how developers navigate and browse documentation artifacts during maintenance tasks. We asked 33 participants to perform 8 different maintenance tasks on a Java software system providing them, besides the source code, use case descriptions, sequence diagrams, class diagrams, and Javadocs. Through an Eclipse plugin, we recorded how much time participants spent on different artifacts, and how they navigated from an artifact to another.

Results of our study indicated that participants spent most of their time on source code when identifying code components impacted by a maintenance activity, while preferring sequence diagrams among the available sources of documentation, followed by class diagrams. Also, they generally started their tasks from source code, or from design documents (84% of cases), then browsing back and forth between source code and either class or sequence diagrams. Less frequently, participants—especially more experienced ones (i.e., graduate students)—-followed an "integrated" approach, by using different kinds of artifacts, namely starting from use cases, then accessing design documents (class and/or sequence diagrams), and finally accessing source code.

As a first direction for future work, we plan to corroborate our results by replicating our study with different participants and systems. Moreover, we plan to conduct other controlled experiments to explicitly investigating possible relationships existing between the way developers use the available documentation and the correctness of the tasks they perform.

VII. Acknowledgement

We would like to thank all the students that participated in our study.

References

[1] S. Abrahão, C. Gravino, E. Insfrán, G. Scanniello, and G. Tortora. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Transaction on Software Engineering*, 39(3):327–342, 2013.

[2] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

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

[5] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical evidence about the UML: a systematic literature review. *Software: Practise and Experience*, 41(4):363–392, 2011.

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

[7] J. A. Cruz-Lemus, M. Genero, M. E. Manso, and M. Piattini. Evaluating the effect of composite states on the understandability of UML statechart diagrams. In *proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS 2005)*. Springer, 2005.

[8] B. de Alwis and G. C. Murphy. Using visual momentum to explain disorientation in the Eclipse IDE. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 51–54, Brighton, UK, 2006. IEEE Computer Society.

[9] R. DeLine, A. Khella, M. Czerwinski, and G. G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the ACM 2005 Symposium on Software Visualization*, pages 183–192, St. Louis, Missouri, USA, 2005. ACM.

[10] W. J. Dzidek, E. Arisholm, and L. C. Briand. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Transaction on Software Engineering*, 34(3):407–432, 2008.

[11] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 341–350, Dubrovnik, Croatia, 2007. ACM.

[12] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[13] Y.-G. Guéhéneuc. TAUPE: towards understanding program comprehension. In *Proceedings of the 2006 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2006), October 16-19, 2006, Toronto, Ontario, Canada*, pages 1–13. IBM, 2006.

[14] S. Jeanmart, Y.-G. Guéhéneuc, H. A. Sahraoui, and N. Habra. Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 69–78, Lake Buena Vista, Florida, USA, 2009.

[15] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, Oregon, USA, 2006. ACM.

[16] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

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

[18] M. C. Otero and J. J. Dolado. An initial experimental assessment of the dynamic modelling in UML. *Empirical Software Engineering*, 7(1):27–47, 2002.

[19] R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2012. ISBN 3-900051-07-0.

[20] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato. How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. *IEEE Trans. Software Eng.*, 36(1):96–118, 2010.

[21] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[22] B. Sharif and J. I. Maletic. An eye tracking study on the effects of layout in understanding the role of design patterns. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–10, Timisoara, Romania, 2010. IEEE Computer Society.

[23] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[24] J. Singer, T. C. Lethbridge, N. G. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research*, page 21, Toronto, Ontario, Canada, 1997. IBM.

[25] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3):183–207, 2000.

[26] 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 *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*, pages 184–191, New York, NY, USA, 2003. ACM Press.

[27] M. Torchiano. Empirical assessment of UML static object diagrams. In *International Workshop on Program Comprehension (IWPC 2004)*, pages 226–229. IEEE Computer Society, 2004.

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

[29] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 39–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[30] S. Yusuf, H. H. Kagdi, and J. I. Maletic. Assessing the comprehension of UML class diagrams via eye tracking. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 113–122, Banff, Alberta, Canada, 2007. IEEE Computer Society.