How Do Professional Developers Comprehend Software?

Tobias Roehm
TU München
Munich, Germany
roehm@cs.tum.edu

Rebecca Tiarks
University of Bremen
Bremen, Germany
beccs@tzi.de

Rainer Koschke
University of Bremen
Bremen, Germany
koschke@tzi.de

Walid Maalej
TU München
Munich, Germany
maalejw@cs.tum.edu

***Abstract*—Research in program comprehension has considerably evolved over the past two decades. However, only little is known about how developers practice program comprehension under time and project pressure, and which methods and tools proposed by researchers are used in industry. This paper reports on an observational study of 28 professional developers from seven companies, investigating how developers comprehend software. In particular we focus on the strategies followed, information needed, and tools used.**

We found that developers put themselves in the role of end users by inspecting user interfaces. They try to avoid program comprehension, and employ recurring, structured comprehension strategies depending on work context. Further, we found that standards and experience facilitate comprehension. Program comprehension was considered a subtask of other maintenance tasks rather than a task by itself. We also found that face-to-face communication is preferred to documentation. Overall, our results show a gap between program comprehension research and practice as we did not observe any use of state of the art comprehension tools and developers seem to be unaware of them. Our findings call for further careful analysis and for reconsidering research agendas.

***Keywords*-program comprehension; empirical studies; software documentation; maintenance; context awareness**

I. Introduction

Program comprehension is an important activity in software maintenance. It consumes about half of the time spent by developers in maintenance as reported by Fjeldstad and Hamlen [5]. According to Singer et al. [22], program comprehension mainly takes place before changing code, because developers must explore source code and other artifacts to find and understand the subset of the code relevant to the intended change. The strategies followed to understand software might vary among developers depending on their personality, experience, skills, and task at hand.

The goal of this study is to review the state of the practice in program comprehension and learn how programmers in industry comprehend programs. We aim at gaining deep insights into program comprehension practice, examining the usage of research results in practice, validating findings of similar studies, and overcoming limitations of earlier studies. Furthermore, hypotheses generated from our observations serve as starting point for a need-driven research agenda.

Earlier studies that empirically survey program comprehension practices suffer from limitations calling for a deeper, up-to-date examination. For example, the studies of Fjeldstad and Hamlen [5] and Singer et al. [22] are rather old. Since their conduction new programming languages like Java and practices like agile development and open source development have become popular. LaToza et al. [10] study developers from a single company and Robillard et al. [17] study only five developers in a lab setting. Our study surveys developers working in different companies of different size using different technologies and has a larger sample size of 28 developers. Further, we follow a different method enabling detailed descriptions of rationale and thoughts behind observed behavior.

The contribution of this paper is threefold. First, it describes in detail strategies followed, information needed, and tools used during program comprehension as well as the rationale behind. Second, we develop a catalogue of hypotheses about program comprehension that can be used to guide further research effort and tool development. Third, we describe how we conducted our study, which consists of observations and interviews. The study design can be reused in other studies to understand developer behavior.

The paper is organized as follows. Section II describes the design of our study. Section III summarizes the findings and presents hypotheses derived from observations. Section IV discusses the implications of our findings and threads to validity. Finally, Section V presents related work and Section VI concludes the paper and sketches future work.

II. Study Design

After presenting the research questions of the study, we detail on the method followed, including participant recruitment, and reliability measures.

A. Research Questions

The goal of this study is to *qualitatively explore* how program comprehension is done in software industry. This includes studying how program comprehension tools are used in practice, generating hypotheses about industrial program comprehension, and testing established theories. We refrain from quantifying certain comprehension aspects as the distribution and characteristics of the whole population is unknown and our sample is relatively small in statistical terms. Further, we do not explicitly aim for new theories.

In order to structure the study and focus our effort, we investigate three main areas: strategies developers follow, information developers interact with or that is missing, and tools used. We formulate the research questions as follows:

- RQ1: Which strategies (including steps and activities) do developers follow to comprehend programs?
- RQ2: Which sources of information do developers use during program comprehension?
- RQ3: Which information is missing?
- RQ4: Which tools do developers use when understanding programs and how?

B. Research Method

Our study is designed to meet two requirements: realism and replication. To meet the requirement of realism, we chose situations representative for realistic program comprehension tasks. We observed developers in their *real* work environment. We did not predefine the tasks the participants worked on. The task was chosen by the participants themselves. We only requested a task that included program comprehension activities and that required the participant to spend about one hour of time in order to ensure that the task was large enough to evolve a program comprehension strategy. To meet the requirement of replication, we prepared material such as questionnaires and a protocol form and describe our study in detail in this section.

Our method includes a combination of observation and interview, with the observation mainly targeting what developers do and the interview mainly targeting the motivation behind developer's actions. This combination was most appropriate to answer our research questions. Automated observations through instrumentation of development environment do not reveal reasons and motivations behind observed behavior. Experiments imply controlling particular variables, assuming they are independent. Finally, surveys or interviews assume that the questions are correct and complete. They can be short in explaining real behavior in detail and answers might deviate from real practices.

1) Observation: During the observation session, we concentrated on *which* and *when* questions like "which activities does a developer perform?". For the observation session, we used the *think-aloud* method, i.e. we asked participants to comment on what they are doing thereby enabling the observers to understand what is going on and get access to thoughts in participants' mind. In case participants stopped talking, we asked questions to start the information flow again. But we took care not to interrupt the workflow of participants and deferred questions calling for long, detailed answers to the subsequent interview. Before starting, we shortly presented the study goals to the participants, its exploratory nature (i.e. no right or wrong behavior was expected), and assured them anonymity and confidentiality.

To document the observation sessions, we created a protocol covering the current time, a description of the participants' actions, and quotes. An excerpt of such a protocol is shown in Table I. When we discovered interesting issues like unclear actions or counterintuitive behavior, we noted them down and discussed them in the subsequent interview.

Table I
PROTOCOL EXCERPT OF PARTICIPANT P5

Daytime	Relative time	Observation/ Quote	Postponed questions
...	...	...	...
10:19	00:27	Read Jira ticket *Comment: "this sounds like the ticket from yesterday"*	*What information considered?*
10:20	00:28	Refresh source code repository	
10:24	00:32	Publish code to local Tomcat	
10:26	00:34	Debug code in local Tomcat	*Why debugging?*
10:28	00:36	Open web application in browser and enter text into form fields	
10:29	00:37	Change configuration in XML file content.xml *Exclamation: "not this complicated xml file again"*	*How known what to change?*
10:30	00:38	Publish changes to local Tomcat	
10:31	00:39	Debug local Tomcat	
...	...	...	...

We observed a single participant for 45 minutes, leaving us another 45 minutes for the interview and not spending more than 1,5 hours in total. The reason for this time constraint was not to observe people too long as concentration decreases over time. In each session, one participant was observed and interviewed by one observer.

2) Interview: During the observation sessions, we got a good understanding of the context of participants and their workflows. In order to gain more insight about rationale behind actions and figure out whether the observed behavior is representative for the developer, we conducted a "contextualized" interview directly after each observation session. We put special emphasis on understanding events and actions that occurred during the observation.

The interview focused on exploring *how* and *why* questions like "Why did you debug?" or "How did you realize that method Y is buggy?". We conducted semi-structured, exploratory interviews. We used prepared questions[1], but did not stick to them rigidly. We rather used the questions as guidance to explore the subfields of program comprehension we were interested in, i.e. strategies employed, information sources, missing information, and tools used. At the start of each interview, we gave a short definition of program comprehension in order to focus the discussion. Minutes were manually created one to two days after each interview.

3) Testing the Method: Before we observed programmers in industry, we conducted a test session with a postgraduate student and observed him during a development task using a first version of our observation sheet and questions. The

[1] http://www.informatik.uni-bremen.de/st/pumba.php?site=interview

observation sheet turned out to be suitable, its use became better with increasing experience of observers. However, the questions had to be revised. First, we realized that we had too many questions and dropped less important ones in order to stick to our timeframe of 45 min. Second, we realized that some questions target the same information and we merged them, for instance, "Which steps are difficult when understanding software?" and "Which problems occurred during understanding software?".

4) Evaluation: In order to analyze the results of observations and interviews we used two different approaches.

- We summarized each session by collecting interesting observations, i.e. those that were not expected, occurred in a similar way in other study sessions, or differed among participants. We clustered these observations by topic and compared similarities and differences in the behavior of different participants with respect to a specific topic.
- We collected all answers to a specific interview question, summarized them, and compared the answers of different participants.

The results of both evaluation approaches were iteratively combined and incorporated in textual descriptions, which can be found in Section III. The validity of a single result is strengthened if it is both observed and reported.

5) Reliability Measures: To increase the reliability of our findings, we employed the following measures [3]:

- Independent peer observations: To eliminate observer bias, we report in this paper only on observations that were independently reported at least in two different sessions by two different observers.
- Peer debriefing: After each study day, every observer discussed its observations and findings of that day with another author. Discussing results helped in summarizing important findings, relating results to results from other sessions, and interpreting observations.
- Triangulation of data sources: Our study design has the advantage that it produces two kinds of data sources: observation protocols containing what observers saw and interview transcripts containing what participants told. Combining these triangulates between different data sources and improves reliability.
- Participant checking: We sent a first draft of the findings in Section III to two participants that were observed earlier and asked them for feedback. Participant P1 agreed with 21 hypotheses, was not sure for Hypothesis 4 and 23. He disagreed with Hypothesis 21 which he could not confirm by experience. Participant P2 agreed with 18 hypotheses, was not sure for Hypothesis 4, 5, 7, 19, and 22, and disagreed with Hypothesis 23 which he could not confirm by experience.

Table III
OBSERVATIONS WITH PARTICIPANT AND COMPANY COUNTS

Observation	# P	# C
Comprehension Strategies		
(S1) Employ a recurring, structured comprehension strategy depending on context	26	7
(S2) Follow a problem-solution-test work pattern	18	5
(S3) Interact with UI to test expected program behavior	17	5
(S4) Debug application to elicit runtime information	16	5
(S5) Clone to avoid comprehension and minimize effort	14	6
(S6) Identify starting point for comprehension and filter irrelevant code based on experience	10	5
(S7) Establish and test hypotheses	9	5
(S8) Take notes to reflect mental model and record knowledge	9	4
Information Sources		
(I1) Source code is more trusted than documentation	21	6
(I2) Communication is preferred over documentation	17	5
(I3) Standards facilitate comprehension	12	6
(I4) Cryptic, meaningless names hamper comprehension	10	6
(I5) Rationale and intended usage is important but rare information	10	5
(I6) Real usage scenarios are useful but rare	5	4
Tool Usage		
(T1) Dedicated program comprehension tools are not used	28	7
(T2) Standalone tools are used in addition to IDEs	5	4
(T3) Compiler is used to elicit structural information	5	4
(T4) Tool features for comprehension are unknown	3	3

C. Participant Recruitment

Our participants had to work for a software development company and spend most of their time coding (to make sure to study software developers). We excluded other people, especially students and university researchers, from the study because we want to study industry practice. We also allowed participants with different tasks, different project roles, different experience, different technology used and different company size in order to explore program comprehension as broad as possible and to improve external validity.

Table II gives an overview of all 28 participants. Five participants work for companies located in Spain and the rest for companies located in Germany. The column *W. exp.* represents how many years of work experience a participant had. The role developer denotes that the participant mainly implements new functionality. The role maintainer denotes that the participant mainly fixes bugs. The column *Fam.* denotes whether a participant was familiar with the program they are trying to comprehend during the observation. *Tec. exp.* represents the experience with the technology used in years. All participants worked in projects that used a proprietary customized agile development process.

III. Findings

We summarize the observation and interview answers, formulate hypotheses summarizing the observations, and relate our findings to the findings of similar studies. Table III gives an overview of our findings and the number of participants and companies that support each finding.

Table II
OVERVIEW OF PARTICIPANTS

ID	Company	W. exp.	Project role	Domain	Fam.	Technology Used	Tec. exp.	Task During Observation
P1	C1 (DE)	4.5	Developer, Maintainer	Facility control		Java, Netbeans	4.5	Application familiarization
P2	C2 (DE)	3	Technical documenter	Fleet management		PL/ SQL, Oracle SQL Developer	0.25	Technical documentation
P3	C3 (ES)	8	Manager, Developer	Event management	X	Delphi, Delphi IDE (Client), Java, Eclipse (Server)	6	Bug fixing
P4	C3 (ES)	8	Manager, Developer	Event management	X	Delphi, Delphi IDE (Client), Java, Eclipse (Server)	6	Feature implementation
P5	C4 (ES)	2	Developer	Port management	X	Oracle DB, Java, Oracle Toad, Eclipse, Tomcat	2	Bug fixing Feature implementation
P6	C4 (ES)	1.5	Maintainer	Port management	X	Oracle DB, Java, Oracle Toad, Eclipse, Tomcat	1	Feature implementation (2x)
P7	C4 (ES)	4.5	Developer, Maintainer	Port management	X	Oracle DB, Java, Oracle Toad, Eclipse	2.5	Porting a feature
P8	C5 (DE)	3	Developer, Consultant	Automotive software	X	C, ASCET, SourceInsight	3	Application familiarization
P9	C5 (DE)	9	Developer	Automotive software	X	C, NotPad ++	9	Version comparison
P10	C5 (DE)	16	Researcher	Automotive software	X	C, ASCET, NotePad++	16	Version comparison
P11	C5 (DE)	6	Developer	Automotive software	X	C, Eclipse	3	Code review
P12	C5 (DE)	8	Developer	Automotive software	X	C, Eclipse, SourceInsight	8	Code review
P13	C5 (DE)	7	Developer	Automotive software	X	C, Visual Studio	7	Feature implementation
P14	C5 (DE)	6	Developer, Maintainer	Automotive software	X	C, XML, CodeWrigth Editor	3	Feature implementation (2x)
P15	C6 (DE)	1.5	Developer	Computer Aided Design	X	Python, Eclipse	1.5	Feature implementation (3x)
P16	C6 (DE)	19	Developer	Computer Aided Design	X	Python, Eclipse	3	Bug fixing
P17	C6 (DE)	5	Developer	Product management	X	Python, SQLite, Toad	5	Code review
P18	C6 (DE)	7.5	Developer	Databases	X	C, Python, XML, Vi	5	Feature implementation Porting a feature
P19	C7 (DE)	3	Developer	Content management	X	C#, Visual Studio	3	Bug fixing
P20	C7 (DE)	11	Developer	Content management	X	VB, VB .NET, Visual Studio	8	Bug fixing
P21	C7 (DE)	11	Developer	Content management	X	VB .NET, C#, Visual Studio	8	Bug fixing
P22	C7 (DE)	16	Developer	Content management	X	Java, Tomcat, NetBeans	11	Feature implementation
P23	C7 (DE)	1.5	Developer	Content management	X	Java, NetBeans	1.5	Feature implementation
P24	C7 (DE)	11	Developer	Content management	X	Java, Eclipse	11	Feature implementation
P25	C7 (DE)	3	Developer	Content management	X	VB .NET, SQL Server, Visual Studio	3	Bug fixing
P26	C7 (DE)	18	Developer	Content management	X	VB, VB .NET, Visual Studio	18	Bug fixing
P27	C7 (DE)	8	Developer	Content management	X	VB. NET, Visual Studio, Editor	4	Bug fixing
P28	C7 (DE)	10	Developer	Content management	X	VB .NET, Visual Studio	5	Bug fixing

A. Comprehension Strategies

By comprehension strategy, we mean the overall approach and activities performed to reach a certain comprehension goal, e.g. debugging to determine why a Nullpointer exception occurs or asking colleagues to acquire a certain information. We observed the following 8 strategies.

(S1) Employ a recurring, structured comprehension strategy depending on context: We noticed that many participants approached tasks using a recurring, structured strategy. 26 participants confirmed in the interviews that they follow such a strategy. But the strategies differed among them. Sixteen participants argued that they start with reading source code and locating the code where the change should be performed. Three participants said that they start with inspecting documentation or requirements. P1 reported that his comprehension strategy depends on the type of application: in case of a server application or library, he tests possible calls and the corresponding behavior (a black box approach) whereas in case of applications with a Graphical User Interface (GUI) he identifies methods that are executed as a consequence of button clicks (a white box approach). The strategy used by Participant P7 depends on previous knowledge. If he already knows an application, he runs it and inspects source code. But, if he is completely unfamiliar with an application, he either talks to a person with knowledge about it or implements some dummy functionality to test the application behavior. P3 reported to use a task independent, high-level strategy by ensuring that code can be compiled and run as a prerequisite for all other comprehension activities. We observed that participants used different strategies for bug fixing (e.g. reproduce bug, locate cause, apply fix) and feature implementation (e.g. understand behavior of application, analyze similar code, copy and adapt code).

Hypothesis 1 Developers usually follow a recurring, structured comprehension strategy that varies with the type of task, developer personality, the amount of previous knowledge about the application and the type of application.

There is a distinction between what we describe as "recurring, structured approach" and what Littman et al. describe as "systematic and opportunistic strategies" [12]. What Littmann et al describe as "systematic and opportunistic" approach refers to reading code line by line (systematic) or in a more arbitrary order. Our strategies are not limited to the way developers read code but describe a whole "workflow" (e.g. reading documentation, locating a bug, applying changes). We did not observe whether the code reading as a part of those strategies was systematic or opportunistic. The strategies we observed were recurring and depend on the context factors mentioned in Hypothesis 1.

(S2) Follow a problem-solution-test work pattern: We observed that 18 participants followed a work pattern including three steps: identifying the problem, implementing the solution, and testing the solution. Each of these steps had a different comprehension goal. The focus of the first step was to understand what happened before a bug occurred and why this causes the bug (in case of bug fixing) or to understand application behavior (in case of feature implementation). The focus of the second step was to understand how the bug can be removed (bug fixing) or how the feature can be implemented, e.g. which code has to be changed and which code has to be added.

For example, P3 tried to fix a coloring bug in the UI. He resumed this task because he could not finish it the day before. P3 tried to identify the cause of the bug by debugging the application and executing it with print statements (step 1). Despite several code modifications (step 2) and subsequent testing (step 3), P3 could not solve the problem during our observation session. In contrast, P5 was able to complete two tasks during our observation session and completed two passes through the work pattern. The first task was to fix a SQLException problem. P5 inspected the problem by analyzing the SQLException trace and discovered that two table attributes were missing in the database of the production environment (step 1). As solution, P5 wrote a SQL script creating those two missing attributes (step 2) and tested it by running it on the local test system (step 3). The second task was to add additional information to a view in the GUI. P5 debugged the application in order to identify the code location where to add new instructions (step 1), copied an existing table structure, adapted it, added code to fill the adapted structure (step 2), and tested the implementation by restarting the application and inspecting the table in the GUI (step 3).

Similarly, Boehm [1] observed that modifying software generally involves three phases: understanding the existing software, modifying the existing software, and revalidating the modified software. Maalej and Happel [14] also found that about 25% of developers describe their work by using problem-solution phrases.

Hypothesis 2 For tasks including changing source code, developers employ a work pattern with three steps: 1) Identification of the problem (in case of bug fixing) or identification of code locations as starting points (in case of feature implementation), 2) searching for and applying a solution, and 3) testing the correctness of the solution.

(S3) Interact with UI to test expected program behavior: 21 participants worked on applications that exhibit a user interface (UI) and 17 of them used the UI to comprehend the application or stated this in the interview. P1 inspected which code is triggered by a button click and used this information as a starting point in exploration. P3 related control flow to the user interface during debugging: "we only passed two times in this loop because we have two categories of events displayed in the user interface". P2, P5, P17 and P19-P28 interacted with the user interface (entered values in text fields, expanded drop down lists, or clicked on buttons) in order to familiarize with the application's functionality and test whether the application works as expected. P4 tested the correctness of application implementation and his conceptualization of it by entering values in a UI form and verifying that these values are stored in the database.

Hypothesis 3 Developers interact with the user interface of the software to test if the application behaves as expected and to find starting points for further inspection.

(S4) Debug application to elicit runtime information: All 21 participants from companies C1-C4, C6 and C7 read source code and executed the application. Sixteen of them used the debugger to inspect the state of the application at runtime. The participants from company C5 did not have the possibility to execute or debug the application because they can only access and compile parts of the software they are working on as the overall system has to run on a special hardware that is not available to single developers.

This finding matches with results of Murphy et al. [15] who found that debuggers are frequently used by developers.

Hypothesis 4 Developers frequently debug the application to acquire runtime information.

(S5) Clone to avoid comprehension and minimize effort: We observed 14 participants reusing code or documentation by cloning. To avoid breaking existing code, P3 copied a piece of code and adapted it instead of refactoring the original code. The fear of breaking originates from not knowing all possible usages of the piece of code ("I do not know if it is used otherwise") and not being able to test all possible usages after a modification to check the correctness of the modification ("I can't test everything later").

P2 stated another reason for copying and adapting documentation: "copy documentation to have a uniform structure", simplifying the use of the documentation and the access of information needed during comprehension.

An interesting code reuse strategy was observed for participant P7. The task was to re-implement an already existing functionality from another customer-specific version of the application. P7 copied big blocks of code containing several methods and "deactivated" the whole copied code by commenting it out. Then he looked at compiler warnings that indicated that a certain method was not found by the compiler. Following these warnings, P7 removed the comments and "reactivated" the appropriate methods. P7 applied this strategy repeatedly until all compiler warnings disappeared. Using this strategy, P7 traced which methods in the copied code were really needed. With this strategy, P7 did not try to understand how each method works.

Another reason reported by participants for cloning code is to reuse an existing implementation and save effort compared to writing code from scratch. Most participants claimed that they clone code to save effort and only few reported that they clone code to avoid comprehension.

This finding matches with the results of Singer et al. [22], who reported that novice developers do not comprehend system aspects that go beyond the needs of their current tasks.

Hypothesis 5 Developers try to avoid comprehension by cloning pieces of code if they cannot comprehend all possible consequences of changes.

Hypothesis 6 Developers prefer a unified documentation structure to simplify finding information needed for comprehension.

Hypothesis 7 Developers usually want to get their tasks done rather than comprehend software.

(S6) Identify starting point for comprehension and filter irrelevant code based on experience: During the observations we noted that many participants had an idea where to start inspecting the program behavior. When asked how they choose these starting points, participants agreed that they know from experience where to begin. For example, P3 explained “if you know the application you know where to touch the code”. Eight participants explained, that they choose a specific starting point “because this is always the starting point in all our systems” and most of them referred to the application architecture as crucial information.
We observed that experience also influenced the recognition of data structures and helped in the location of concepts. P11, for example, realized very quickly that a specific feature in the software was implemented as a state machine because he knew this concept from other parts of the system.

We also observed that participants often decided whether to take a closer look on a certain code fragment or not based on experience. Participants P2, P9 and P10 ignored parts of the source code, arguing e.g. that “this part implements calculation, which is not important for my current task”.

This observation matches with results from Sillito et al. [19] who found that developers minimize the amount of code to read.

Hypothesis 8 Experience of developers plays an important role in program comprehension activities and helps to identify starting points for further inspection and to filter out code locations that are irrelevant for the current task.

(S7) Establish and test hypotheses: We observed that participants comprehend code by asking and answering questions or establishing hypotheses and testing them. Nine participants verbalized such questions or hypotheses during observations. For example, P1 asked questions like “Where do Corba calls happen?” or “What do I have to change to implement user profiles?”. Four participants (P11, P12, P14, P15) asked themselves where certain values were set or used. Participants P2 and P3 established hypotheses about the application behavior and compared them to actual observed behavior, reflected in statements like “I assume this method fetches the maximum value” (P2), or “the problem should be in method prepareItem” (P3), or “the values printed should be all x ... [printed values differ from x] ... oh, they are different” (P3).

This finding matches with results reported by Brooks [2], Mayrhauser et al. [24], and Ko et al. [8]. According to these authors, questions lead to informal hypotheses that are verified by developers.

Hypothesis 9 Developers comprehend software by asking and answering questions and establishing and testing hypotheses about application behavior.

(S8) Take notes to reflect mental model and record knowledge: Nine participants took notes on a separate piece of paper or used a text editor as a temporal memory during comprehension activities. These notes varied from single function names, mappings between ids and labels to complex flow charts corresponding to the part of the application studied. We also observed one participant writing down how a specific module could be used and accessed. Three participants drew flow charts. They started by writing down a condition that they assumed to be the starting point. During the next inspection, the participants refined the charts by adding further conditions and incorporated additionally acquired knowledge into them. P3 wrote down the method name and parameters of a server call to be able to debug a server call with the original parameters.

All participants started their tasks without using existing notes. P9 stated that “notes are only important for the current mental model” and participant P11 reported that notes “are only for personal understanding” and are not archived or used beyond the current task.

Hypothesis 10 Some developers use temporal notes as comprehension support. This externalized knowledge is only used personally. It is neither archived nor reused.

B. Information Sources

We made six main observations about information sources needed during program comprehension, how knowledge was documented and shared, and which information was missing to developers.

(I1) Source code is more trusted than documentation: 21 participants reported that they get their main information from source code and inline comments whereas only four stated that documentation is their main source of information. P2 verified the correctness of existing documentation by inspecting source code in order to make sure that "a ratio in the documentation is really a ratio". P9 questioned the trustfulness of documentation in general ("you cannot trust the documentation") and P5 reported that "technical documentation covers only 10 % of the application". The lack of documentation was confirmed by P1: "source code is documented sparsely". P5 explained that the reason for this phenomenon as follows: "Documentation costs much time, usually more than actual implementation. That's the reason why people try to avoid documentation."

This finding matches with the results of other researchers, who reported that documentation is seldom kept up-to-date [6], [10], [11]. Singer et al. [22] also reported that source code is read frequently while documentation is not.

Hypothesis 11 Source code is considered a more credible source of information than written documentation, mainly because documentation is often non-existent or outdated.

(I2) Communication is preferred over documentation: Seventeen participants reported that communication with colleagues is a more important source of information than written documentation. P5 reported that "only little is documented, most knowledge is in comments or experts' heads" and P1 employed a strategy to ask colleagues for information in case of problems in components developed by them. Participants from company C7 stated that communication is the most important source of information as documentation is rarely available. P4 and P7 compared the benefits of writing documentation with explaining to colleagues orally. According to them, an explanation can be tailored to the information seeker by relevance or by previous experience, whereas a written documentation can be re-read in case some details were forgotten. P7 stated "I have the feeling that our bosses want us to explain to people, not to document". P15 and P17 stated that "due to the small size of the project team it is much easier to go next door and ask a colleague than searching for the information needed in the documentation".

This result is consistent to the findings reported by LaToza et al. [10], who emphasized the importance of people to gain information compared to written documents.

Hypothesis 12 Communication with colleagues is a more important source of information than written documentation because written documentation is non-existent and some developers prefer direct communication over writing documentation. The advantage of direct communication is that answers can be tailored to the information seekers whereas the advantage of written documentation it is reusable.

(I3) Standards facilitate comprehension: Twelve participants agreed that the consistent use of naming conventions and a common architecture simplify program comprehension tasks considerably. P1 and P3 searched for starting points to inspect the code by using the standard structure of an application. P1 used web application resources such as the web descriptor web.xml, while P3 used standard naming scheme of Delphi UI triggers such as SHOW or CREATE methods. P5, P6, and P7 reported that a standardized architecture helps them to locate code that has to be changed for a bug fix or a feature implementation. Further, P2 reported that most of the functions he analyzed had the same structure ("filter data, set global variables, calculation") and he ignored those parts of the structure that are not relevant for his current task. Naming conventions played a central role for company C5. The participants even used a self-developed translator that transformed cryptic names of functions and variables into a meaningful human readable form, which helped to get a better understanding of the software. This finding is consistent with the finding of Rajlich and Wilde [16], who reported that "regularities in the design, and especially in the naming of functions and data, may greatly facilitate concept location".

Hypothesis 13 Standardization – the consistent use of naming conventions and a common architecture – allows developers to become familiar with an application quickly and makes program comprehension activities easier and faster.

(I4) Cryptic, meaningless names hamper comprehension: The issue of low quality names of variables, methods, and constants was raised in the observations and interviews by ten participants. P2 was angry when encountering cryptic variable names like CT_AVG_AC or GT_CCMP several times and had no idea what they meant. P6 explained that a reason for using cryptic names is that database field names were restricted to 5 characters and hence only abbreviations could be used as names. P3 reported that confusing names are due the lack of a mandatory coding style and the mixture of English and Spanish names. Additionally, the participant stressed the importance of semantic names of trigger components like SHOW to identify source code with a specific functionality. P5 explained that "well written code uses semantic names" and "50 % of code is well written". During the interviews, two participants explicitly said that it is very difficult to understand source code that is not properly formatted according to style guides. Company C5 enforces naming conventions that can be translated to semantic names by a translator. Participants from C5 agreed that this helps to understand the rationale behind the code. However, improper use of naming conventions also led to misunderstandings. P9, for example, assumed a different meaning of a function due to its misleading name. He explained that "the function name took me to the wrong direction. According to our naming convention the name should have been different".

Hypothesis 14 Cryptic, non-semantic names hamper understanding of a piece of code.

Hypothesis 15 Naming conventions can help to mitigate this effect but if they are too complicated they can have a negative effect.

(I5) Rationale and intended usage is important but rare information: Participants were interested in information about the "purpose and idea behind a class or method" and "how it should be used". P1 mentioned that rationale gets lost and cannot be restored when it is not documented, even for code written by the participant himself ("without documentation I would forget quickly"). The results from the observation were approved during the interview where ten participants argued that understanding the rationale behind the code is very exhausting. In contrast, we did not observe a single participant documenting rationale for own code.

This finding supports the result of LaToza et al. [10] that understanding the rationale behind code is a big problem for developers.

Hypothesis 16 Knowledge about rationale of the implementor and intended ways of using a piece of code help to comprehend code but this information is rarely documented.

Hypothesis 17 There is a gap between the interest of developers in this information and the lack of documenting it for their own code.

(I6) Real usage scenarios are useful but rare: Five participants reported the importance of knowledge about how end users use the application as context information for comprehension. P3 explained that "needs that are supported by the application" are an information necessary to understand programs. P2 had a dedicated item in his documentation scheme called "user goals". He reported that use cases and requirements of potential users is an information that is missing to him ("I am not sure how potential users will use the system or what their intentions are"). P15 reported that it is often unclear how the end user uses the application and that he has limited knowledge about the application domain.

Hypothesis 18 The way in which end users use an application is a helpful context information in program comprehension.

Hypothesis 19 In many cases this information is missing.

C. Tool Usage

We made four main observations about tool usage, i.e. which tools were used and how they were employed to understand software.

(T1) Dedicated program comprehension tools are not used: Twenty two participants used an Integrated Development Environment (IDE) to read source code and sixteen participants used the debugger to inspect the state of the application during its execution. But we did not observe any usage of special program comprehension tools such as visualization, concept location, or software metric tools. Some participants from company C5 used SOURCEINSIGHT[2] to view source code but we did not observe them utilizing the built-in program comprehension features. P3 reported to use WIRESHARK[3], a network packet analyzer, to inspect network communication, while P4 reported to use SELENIUM[4], a macro recorder of user interactions, to simulate user interactions for testing purposes.

Hypothesis 20 Industry developers do not use dedicated program comprehension tools developed by the research community.

(T2) Standalone tools are used in addition to IDEs: During the observations we realized that even though 22 participants used an IDE to read and change source code, five participants employed other tools to perform actions that could also be done by the IDE. For example, P14 used the ECLIPSE IDE that supports full-text search but the participant executed a search using GREP from the command line. P14 argued that "it is much quicker and I am more used to this kind of search". P3 viewed a previous source code version in NOTEPAD and the current version in DELPHI because "it would be too difficult to open this also in DELPHI and switch between the current and previous code versions".

This finding is consistent with the results from LaToza et al. [10], who reported on the use of standalone tools in addition to an IDE in a bug fixing scenario. Singer et al. [22] and Maalej [13] also found that developers complain about loose integration of tools, which might hinder comprehension as information required is scattered across different tools.

Hypothesis 21 During comprehension tasks, IDE and specialized tools are used in parallel by developers, despite the fact that the IDE provides similar features.

(T3) Compiler is used to elicit structural information: Five participants used the compiler to elicit structural information. P3 used the compiler to search for locations where a specific constant is used by changing the name of that constant in its definition and examining the locations of the resulting compiler errors. P4 used the compiler to find out where he inconsistently adapted a copied piece of code, e.g. inconsistent changes of variable names. P7 used the compiler extensively to check which methods of a copied code block are necessary as described in finding S5. P15 and P18 used the compiler messages to find error locations. Starting from the compiler message "variable x not defined" they performed a full-text search for the code locations where variable x is used.

Hypothesis 22 The compiler is used by some developers to elicit structural information such as dependencies and usage locations of code elements.

(T4) Tool features for comprehension are unknown: We made an unexpected observation with P3. In order to find all

[2]http://www.sourceinsight.com/
[3]http://www.wireshark.org/
[4]http://seleniumhq.org/

code locations where a specific constant is used, P3 changed the name of the constant in its definition and inspected resulting compiler warnings. P3 was working in ECLIPSE that provides the feature REFERENCES for retrieving such a list of constant usage. When asked about the motivation behind this behavior, P3 answered that he did not know the ECLIPSE feature despite of 8 years of professional experience and 6 years using ECLIPSE. P22 solved the same problem by performing a full text search for a method name. These observations match with results from Sillito et al. [19], who reported that developers make inefficient use of tools.
Hypothesis 23 Developers do not know some standard features of tools.

IV. DISCUSSION

We discuss the implications of our findings for researchers, tool vendors, and practitioners and reflect on limitations and threads to validity.

A. Implications

The fact that none of the participants used dedicated program comprehension tools such as visualization, concept location, or software metric tools reveals a gap between program comprehension research and practice.

1) Implications for Researchers: In order to validate, understand, and deal with the gap between research and practice, researchers should investigate the scope of the gap, reasons behind it, and align research efforts to the needs of industry. Possible reasons behind the gap are a) research results and their benefits being too abstract for industry, b) lack of knowledge about available tools among practitioners, c) fear of familiarization effort and lack or trust in new tools, or d) that using new tools requires too much training for practitioners.

2) Implications for Tool Vendors: Similar to researchers, vendors of software development tools should investigate the reasons behind the gap between research and practice, carefully select which tools developed by the research community that provide benefits for program comprehension tasks and incorporate them as features in their tools.

The observation that some developers do not know standard features such as the ECLIPSE feature REFERENCES (see Hypothesis 23) emphasizes the need to educate developers to use tools efficiently and to proactively inform them about new features. We wonder: how many features can be incorporated into a tool such as ECLIPSE before it becomes to overwhelming for a developer to first comprehend its features and then use them appropriately in program comprehension.

Moreover, insights about the work patterns of developers from this and other studies (e.g. [15]) can be used to build tools that are aligned to the workflow of developers. For example a tool, that detects the current high-level activity of developers from low-level actions, can be used to provide only information that is relevant for the current problem [18].

3) Implications for Practitioners: In order to benefit from research results on program comprehension, practitioners should examine the results of the research community, assess their usefulness, and ask tool vendors to incorporate them in their tools.

As standardized coding style was reported to facilitate (see Hypothesis 13) and cryptic names to hinder (see Hypothesis 14) program comprehension, practitioners should think about what kind of coding style they want to implement in their organization if not yet done. If a coding style already exists, it may be worth to analyze its impact on comprehension.

The observation that communication is preferred over documentation in many comprehension situations (see Hypothesis 12) has two drawbacks. First, information might get lost when experts leave the organization. Second, experts frequently get interrupted from their tasks. Practitioners need thus to assess this tradeoff when deciding about information sources to use for program comprehension.

B. Limitations and Threats to Validity

There are several limitations to the internal and external validity of our results. As for the internal validity, we are aware that in 45 min., we can only observe a fraction of developer's work day. We might have missed certain types of tasks, comprehension strategies, information sources, or tools. However, we think that extending the observation time would not fundamentally change the findings, due to the variation of the tasks observed. First, these tasks were randomly selected by the participants. We only constrained the tasks to include program comprehension. Second, the tasks observed were different in duration and nature. While few subjects managed to complete two tasks in the observations session, others did not manage to complete one task.

Another potential threat to the internal validity is that observers might had assumptions and expectations and might considered only clues affirming these expectations, while ignoring clues indicating different unexpected behavior (observer bias). In order to deal with this threat, we report in this paper only on findings that were observed by two observers independently from each other in two different sessions. Similarly, participants might have behaved differently because they were observed. This threat cannot be eliminated completely but we addressed it by assuring participants complete anonymity and confidentiality. We also stressed that there was no "right and wrong behavior" as we only aimed at documenting the state of practice.

Finally, there might be misinterpretations of the think aloud comments and interview answers due to insufficient language skills. For 26 sessions, both participant and observer were either native or proficient speakers. In two sessions, a translator was present due to participant's insufficient English skills. Keeping in mind our reliability measures (i.e. the triangulation of data sources and the participant checking), we think that the effect of this threat is minimal.

Our study was designed to have a strong degree of realism rather than a high external validity. Because we did not study a random sample that is fully representative of the target population of software developers, it is difficult to generalize our findings. In addition, we neglected other interesting aspects such as time spent on single activities, or communication behavior within a team.

We were unable to draw representative samples from all developers of the companies involved. However, the distribution of participants includes different company sizes, different experiences, different application domains, different programming languages, different roles, and different countries – representing a wide range of potential participants. This give use some confidence that the results have a medium degree of generalizability.

V. Related Work

Singer et al. [22] empirically studied developer's habits, and tool usage during software development. LaToza et al. [10] conducted a similar study focusing on software maintenance. Both studies were hosted in a single company, while our sample includes developers from various companies and domains.

Other researchers studied developer behavior during maintenance tasks and program comprehension activities. DeLine et al. [4], Sillito et al. [19], Ko et al. [9], and Robillard et al. [17] studied developer behavior while updating unfamiliar source code. These studies used an experimental setting with a small number of participants from one or two companies working on unfamiliar code. Von Mayrhauser et al. [23], [24] also studied developer's behavior during maintenance tasks, focussing on cognitive processes and mental models. Their results are complementary to ours, as we focussed on other research questions: developer's externalized behavior and its rationale.

Other studies examined information needed to comprehend software. Sillito et al. [20], [21] examined what information developers seek during software maintenance tasks. Ko et al. [7] studied information needs in software development in general. Both studies concentrated on information needs formulated in form of questions asked and answered during a programming task. We also observed information needed focussing on the sources of information and overall access and sharing approaches. These studies observe developers from a single company.

Lethbridge et al. [11] and Forward and Lethbridge [6] studied how developers use and maintain documentation. We made similar observation without restricting our observations to software documentation but also comprehension approaches and tool usage.

Other studies examined how developers use tools to accomplish maintenance tasks. Murphy et al. [15] studied how programmers use the ECLIPSE IDE by analyzing the interaction data collected by instrumenting ECLIPSE. Maalej [13] studied tool usage and problems developers face regarding tool integration using interviews and online questionnaires. These studies used different research methods and are thus complementary to our study. We explicitly identified overlaps between results of related studies and our study during the description of our findings.

VI. Conclusion

In this study, we observed 28 developers from software industry to get insights into the state of the practice in program comprehension. Some of our findings confirm observations made by other researchers, others are new and surprising. We showed that previous observations (cf. findings S2, S4, S5, S6, S7, I1, I2, I3, I5, T2) are valid in different contexts.

One of our most interesting findings is that developers put themselves in the role of end users whenever possible. We observed developers inspecting the behavior visible in user interfaces and comparing it to the expected behavior. This strategy aims at understanding program behavior and getting first hints for further program exploration. It presents an alternative to reading source code and debugging.

Moreover, developers sometimes try to avoid comprehending programs. Instead, they clone source code and adapt it to fulfill their current task. Cloning avoids comprehending possible consequences of modifying code directly. Wherever possible, developers seem to prefer strategies that avoid comprehension, because of time and mental effort needed. Program comprehension is rather considered as a necessary step to accomplish different maintenance tasks than a goal by itself. When software architecture and code has to be investigated, we found that standards as well as experience are important facilitators to quickly familiarize with an unknown program and find starting points for further investigation.

Most observed developers choose from a set of structured comprehension strategies (e.g. follow a problem-solution-test work pattern), depending on their work context. Thereby, context constitutes of the type of task at hand, the type of program to comprehend, previous knowledge about the program, and the developer's general experience.

Overall, we found that state of the art tools in program comprehension are either unknown or rarely utilized. This reveals a gap between the state of the art of program comprehension research and the state of practice in industry (at least in observed companies). The next step is to investigate the reasons behind this gap and to test the 23 hypotheses resulting from our study to improve generalizability.

Acknowledgements

We are very grateful to all study participants. This work was supported by the Deutsche Forschungsgemeinschaft (grant KO 2342/3-1/BR 2906/1-1) and by the European Commission (FastFix project, grant FP7-258109).

References

[1] B. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, 1976.

[2] R. E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[3] J. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage Publications, Inc., 2009.

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

[5] R. Fjeldstad and W. Hamlen. Application program maintenance study: Report to our respondents. In *GUIDE 48*, Philadelphia, PA, 1979.

[6] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, pages 26–33. ACM, 2002.

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

[8] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'04*, pages 151–158. ACM, 2004.

[9] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32:971–987, 2006.

[10] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering, ICSE'06*, pages 492–501. ACM, 2006.

[11] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[12] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, 1987.

[13] W. Maalej. Task-first or context-first? tool integration revisited. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE'09*, pages 344–355. IEEE Computer Society, 2009.

[14] W. Maalej and H.-J. Happel. Can development work describe itself? In *7th IEEE International Working Conference on Mining Software Repositories, MSR '10.*, pages 191–200. IEEE, 2010.

[15] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[16] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 271 – 278, 2002.

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

[18] T. Roehm and W. Maalej. Automatically detecting developer activities and problems in software development work: Nier track. In *Proceedings of the 34th International Conference on Software Engineering, ICSE'12*, 2012.

[19] J. Sillito, K. De Voider, B. Fisher, and G. Murphy. Managing software change tasks: An exploratory study. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering, ISESE'05*, page 10, 2005.

[20] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'06*, pages 23–34. ACM, 2006.

[21] 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.

[22] J. Singer, T. Lethbridge, N. 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, CASCON'97*, page 21. IBM Press, 1997.

[23] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, 1996.

[24] A. von Mayrhauser, A. M. Vans, and S. Lang. Program comprehension and enhancement of software. In *Proceedings of the IFIP World Computing Congress - Information Technology and Knowledge Engineering*, 1998.