X

On the Comprehension of Program Comprehension

WALID MAALEJ, University of Hamburg
REBECCA TIARKS, University of Hamburg
TOBIAS ROEHM, Technische Universität München
RAINER KOSCHKE, University of Bremen

Research in program comprehension has considerably evolved over the past decades. However, only little is known about how developers practice program comprehension in their daily work. This paper reports on qualitative and quantitative research to comprehend the strategies, tools, and knowledge used for program comprehension. We observed 28 professional developers focusing on their comprehension behavior, strategies followed, and tools used. In an online survey with 1,477 respondents, we analyzed the importance of certain types of knowledge for comprehension and where developers typically access and share this knowledge.

We found that developers follow pragmatic comprehension strategies depending on context. They try to avoid comprehension whenever possible and often put themselves in the role of users by inspecting graphical interfaces. Participants confirmed that standards, experience, and personal communication facilitate comprehension. The team size, its distribution, and open source experience influence their knowledge sharing and access behavior. While face-to-face communication is preferred to access knowledge, knowledge is frequently shared in informal comments.

Our results reveal a gap between research and practice as we did not observe any use of comprehension tools and developers seem to be unaware of them. Overall, our findings call for reconsidering the research agendas towards a context-aware tool support.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Documentation, Human Factors

Additional Key Words and Phrases: Empirical Software Engineering, Context Aware Software Engineering, Knowledge Sharing, Program Comprehension, Information Needs

ACM Reference Format:
Maalej, W., Tiarks, R., Roehm, T., Koschke, R. 2013. On the Comprehension of Program Comprehension. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article X (March 2014), 38 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Program comprehension is the activity of *understanding* how a software system or a part of it works. Fjeldstad and Hamlen [1979] reported that developers spend about half of their time to comprehend programs during software maintenance. According to Singer et al. [1997], program comprehension mainly takes place before changing

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).
Author's addresses: W. Maalej and R. Tiarks, University of Hamburg, Department of Informatics; T. Roehm, Faculty of Informatics, Technische Universität München; R. Koschke, University of Bremen, Department for Mathematics and Computer Science.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1539-9087/2014/03-ARTX $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

code, because developers have to explore source code and other artifacts to identify 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, task at hand, or technology used.

Program comprehension is a knowledge-intensive activity, in which developers consume and produce a significant amount of knowledge about software. Previous studies found that developers spend up to 50 percent of their time searching for information [Ko et al. 2007; Murphy et al. 2006] to answer their questions about the system under development. On the other hand, they also produce knowledge about the software, which can answer others' questions [Mark Ackerman and Volker Wulf 2003; LaToza et al. 2006]. For instance, during an error handling task, developers might need clarifications such as which part of the system is concerned by the error or who had experience with the error message [Breu et al. 2010]. They might ask colleagues or search the web to find out how others solved the error.

This research reviews the state of the *practice* in software comprehension by using a combination of qualitative and quantitative research methods. Our goal is to learn *how* developers in industry comprehend software and how they access and share knowledge about it. We aim at gaining deep insights into program comprehension practice and at examining the impact of research results in practice. By overcoming the limitations of earlier studies, we validate similar findings and identify new trends, which can serve as starting point for a need-driven research agenda.

Earlier studies that empirically surveyed program comprehension practices suffer from limitations calling for a deeper, up-to-date examination. For example, the studies of Fjeldstad and Hamlen [1979] and Singer et al. [1997] are rather old. Since their conduction new programming languages like Java and practices like agile and open-source development have become popular. LaToza et al. [2006] studied developers from a single company and Robillard et al. [2004] studied only five developers in a lab setting. Our research is based on a larger sample size and surveys developers working in different companies of different sizes using different technologies. Further, we follow a different method enabling detailed descriptions of rationale and thoughts behind observed behavior.

Moreover, previous studies of developers' knowledge access and sharing behaviors are mainly based on qualitative research methods and were also conducted within single organizations (e.g., Ko et al. [2007] and Sillito et al. [2008]). There has been no broad, multi-organizational quantitative survey to validate prior findings and put them into specific project settings such as the team size, distribution, and open-source experience. With a total of 1,477 complete survey responses, our research quantifies the popularity of specific knowledge to comprehend programs as well as the channels preferred to access and share this knowledge in practice.

The contribution of this paper is fourfold. First, it describes in detail *strategies followed, tools used, and knowledge needed* during program comprehension as well as the rationale behind. Second, it provides an *estimation for the popularity* of (a) certain types of program comprehension *knowledge*, (b) *channels* used to access and share it, and (c) *problems* encountered when exchanging it. Third, the paper includes a *catalogue of findings* about program comprehension and knowledge needs that can be used to guide further research effort and tool development. Finally, the detailed *description on how we designed* our observational study and our templates used for the observations, interviews, and survey can be re-used in similar studies within or across organizations to understand developers' behavior and measure their assessments.

The paper is organized as follows. Section 2 describes the design of our studies including the qualitative and quantitative parts. Section 3 summarizes the qualitative findings from the observational study focusing on comprehension strategies and com-

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Table I. Research Methods Used in the Studies

Study	Method	Type	Questions types	Focus
Observational study	Observation	Qualitative	Which, When, How	What developers do
	Interview	Qualitative	Why, How, What	Motivation, rationale behind actions
Online survey	Survey	Quantitative	Who, When, How often	Developers assessments, quantification, and generalization

prehension tools. Section 4 summarizes the survey findings quantifying on knowledge needs, channels used to access and share knowledge, and problems encountered. Section 5 discusses the implications of our findings for researchers, tool vendors, and practitioners. Section 6 summarizes the threats to validity, while Section 7 presents related work. Finally, Section 8 summarizes the paper contributions, findings, and their implications.

2. RESEARCH DESIGN

After introducing the overall research questions, we report on the design of the studies, the methods followed, the reliability and validity measures taken when collecting and analyzing the research data, and the recruitment of the participants.

2.1. Research Questions

The goal of this study is to *explore* how program comprehension is done in software industry and to *assess* which knowledge is particularly important for comprehending programs, and how this knowledge is supplied in practice. We aim at testing established theories as well as generating findings and hypotheses about industrial program comprehension.

Our study covers three aspects of program comprehension: strategies, tools, and knowledge. By *strategy* we mean the overall comprehension approach, steps, and activities performed to reach a certain comprehension goal. By *tools* we mean the software applications or features of integrated development environments developers use during their comprehension tasks. Finally, by *knowledge* we mean useful information that is either required to comprehend software or generated during the comprehension task. Our specific research questions on comprehension strategies, tools, and knowledge are as follows:

— RQ1: Which *strategies* do developers follow to comprehend programs?
— RQ2: Which *tools* do developers use when understanding programs and how?
— RQ3: Which *knowledge* is important for developers during comprehension tasks?
— RQ4: Which *channels* do developers prefer to access and share knowledge about software?
— RQ5: Which *problems* are frequently encountered by developers while exchanging knowledge about software?

2.2. Research Methods

To answer the research questions we conducted two studies: an *observational study* and an *online survey*. We complementarily used qualitative and quantitative research methods, capturing both the objective behavior and subjective opinion of developers. Table I provides an overview about the methods used in the studies and their focuses.

The observational study had a qualitative focus. We first observed developers in their work environments with a minimal interference and then interviewed them. The observation targeted mainly what developers do and the interview mainly the motivation and reasons behind developers' actions. This combination was most appropriate to answer our research questions RQ1, RQ2, and RQ3.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

We decided for data collection through a human observer rather than automated observation. Automated observations through instrumentation of the development environment do not reveal reasons and motivations behind observed behavior. We did not run the study as a controlled experiment, as our goal was to explore and understand program comprehension rather than to test a particular hypothesis about it. We used a combination of observation and interviews because interviews alone may be misleading in explaining real behavior as answers might deviate from real practices.

One major drawback of human observations and interviews is scalability. Therefore, we complemented our qualitative study with an online survey, which enables quantifying the opinion of a large sample about specific issues. Surveys also allow for the analysis of relationships and for distinguishing between different groups inside the population of developers. We used the survey to measure the assessments of the developers themselves, addressing the research questions RQ3, RQ4, and RQ5. We correlated the answers of developers to identify particular trends amongst them. This allowed for identifying patterns of "developers who said this also said that".

2.2.1. Observational Study. The *observation* session started by a short presentation of the study goals to the participants. We also assured the exploratory nature of the study (i.e., no right or wrong behavior was expected) and the anonymity and confidentiality of the data collected. For the actual observation, we used the *think-aloud* method Ericsson and Simon [1993]. That is, we asked participants to comment on what they are doing thereby enabling the observer to understand what is going on and get access to thoughts in participants' mind. In case participants stopped talking, we asked questions to restart the information flow again. We concentrated on *which* and *when* questions such as "which activities does a developer perform?" We took care not to interrupt the workflow of participants as much as possible and deferred questions calling for long, detailed answers to the subsequent interview.

To document the observation sessions, we recorded field notes of the participants' actions, quotes, and their time of occurrence in an *observation protocol*. An excerpt of such a protocol is shown in Table II. When discovering interesting issues such as unclear actions or counterintuitive behavior, we noted them down and discussed them in the subsequent interview.

In order to gain more insight about rationale behind actions and figure out whether the observed behavior is representative for the developers, we conducted a contextualized *interview* directly after each observation session. The interview focused on exploring *how* and *why* questions like "Why did you debug?" or "How did you realize that method Y is buggy?"

The interviews were semi-structured and exploratory. We used prepared questions[1], but did not stick to them rigidly. The questions served as a guidance to explore the subfields of program comprehension we were interested in, that is, strategies employed, knowledge sources, missing knowledge, and tools used. In order to focus the discussion we gave a short informal definition of program comprehension at the start of each interview. We introduced program comprehension as the task of trying to understand a program, a software system, or a part of it, its behavior, its structure, and how it works.

We noted the answers of the participants in writing using the prepared question templates and then created the formal interview transcripts one or two days after each interview. We refrained from using voice recordings since we found from previous studies that subjects are typically less spontaneous and more cautious when they are

[1]https://sites.google.com/site/pungaproject/templates

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Table II. Excerpt from the Observation Protocol of Participant P5 (Observational Study)

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	
...	...	...	...

recorded. Moreover, the analysis of voice recordings is typically more difficult and time consuming than written transcripts.

A single observation session lasted for 45 minutes, leaving another 45 minutes for the interview. We did not want to spend more than 90 minutes because concentration of both observed developer and observer decreases over time. In each session, one participant was observed and interviewed by one observer.

2.2.2. Online Survey. The survey focused on *knowledge* consumed and produced in software comprehension. Starting from the findings of several recent studies [Ko et al. 2007; Sillito et al. 2008; Fritz and Murphy 2010], we assumed that *knowledge needs* vary with the development task, such as developing new code, understanding legacy code, reusing components, or fixing a bug. Therefore, we first asked developers about their knowledge needs in particular situations. Moreover, there are several ways to access and share this knowledge, for example, via personal communication or Internet forums. We therefore studied which *channels* are used to access and share knowledge about software. Finally, developers might face *problems* to access the knowledge needed to comprehend software, for instance, searching and identifying information might be inefficient or the accessed information incomplete or incorrect. Therefore, we studied the frequency of particular problems during accessing and sharing knowledge.

In the survey, we were interested in the subjective assessments of the importance of particular knowledge needs, channels, and problems based on the experience of developers. Since different people might have different definitions for importance, we decided to measure the importance by the frequency of occurrence. We used a semantic differential scale [Rosnow and Rosenthal 2007] to assess frequencies of knowledge needs, channels usage, and problems faced. Respondents could choose one of the following scale options: never/rarely, seldom/monthly, often/weekly, usually/daily, or "I don't know". Figure 1 shows an example of the questions and the scale used.

When I am trying to <u>understand other's code</u> I need to know...	Never/ Rarely	Seldom ≈ monthly	Often ≈ weekly	Usually ≈ daily	I don't know
What was the coder's intention as he wrote this	○	○	○	○	○

Fig. 1. Example of closed questions and semantic scale used in the online survey.

In total, the survey had 19 questions with fixed-choice items and took about 15 minutes to answer. The survey form was available in English and German[2]. We invite

[2]*The survey form and data can be downloaded from http://www.teamweaver.org/KOMA_Survey/*

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Table III. Validity and Reliability Measures Used in the Studies

	Validity: measure the right thing and reduce risk for assumptions	**Reliability**: measure correctly and reduce systematic errors
Observational study	Realism in subject selection Realism in situation selection Iterative, incremental data aggregation	Pilot testing Use of templates for data collection Independent *peer* observation and peer debriefing Triangulation of sources Participant check
Online survey	Questions based on previous findings Feedback of experts and practitioners Optional free text answers Pairwise statistical tests Open data and forms for replications	Pilot testing Multiple channels to recruit subjects Multiple languages Explanations and contact options Removal of incomplete answers

the reader to use the form for surveying the knowledge exchange practices in their organizations and comparing the results with ours.

2.3. Reliability and Validity Measures During Data Collection and Analysis

To increase the validity and reliability of our research [Rosnow and Rosenthal 2007] we undertook several measures during the data collection and data analysis, which we summarize in Table III.

2.3.1. Observational Study. To meet the requirement of realism [Sjøberg et al. 2003] during the data collection phase, we chose situations representative for realistic program comprehension tasks. We observed and surveyed *real* developers in their *real* work environment. We did not predefine the tasks the participants worked on during the observations. The task was chosen by the participants themselves. We requested only tasks that included program comprehension and that required the participant to spend about one hour of time in order to ensure that a task was large enough to exercise a program comprehension strategy.

To meet the requirement of replication, we prepared a questionnaire, templates for the observation protocols and the interview transcripts. Before we observed programmers in industry, we conducted a *pilot testing* with a postgraduate student and observed him during a development task using a first version of our observation protocol form and transcript template. The observation protocol form turned out to be suitable and its use became better with the increasing experience of the observers. However, the interview 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 during program understanding?” and “Which problems occurred during understanding software?”.

Further, we employed the following measures:

— Independent peer observations [Rosnow and Rosenthal 2007]: To eliminate observer bias, we report only on observations that were *independently reported* at least in two different sessions by two different observers.
— Peer debriefing [Creswell 2009]: After each study day, each of the two observers discussed his or her observations and findings of that day with another author. Discussing results helped in re-estimating overemphasized or underemphasized observations, detailing vague descriptions, relating results to results from other sessions, and interpreting observations. Each debriefing session lasted for about one hour.
— Triangulation of sources [Creswell 2009]: 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

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

triangulates between different data sources improves reliability of the findings. The validity of a single result is strengthened if it is both observed in the observation session and reported in the subsequent interview.

— Participant check [Creswell 2009] (also known as member check): We sent a draft of the derived findings (see Section 3) to five participants and asked them for feedback. Each finding reported in the paper was confirmed by at least one of these participants, while most findings were confirmed by all five participants. The least supported findings with which four of the five participants have disagreed were Finding 12 and Finding 14.

Finally, we followed an *iterative incremental process* to analyze the data and aggregated the observations into findings. After collecting the data, each “result” was documented as a single entry in the observation protocol or the interview transcript. In total we collected 2,807 single observations in the observation protocols (between 120 and 180 observations per session) and 110 pages of interview transcripts. We then grouped the results from each session (observation and interview) into four categories following our research goals: strategies, tools, knowledge, and others. If two researchers agreed that the observation was "not interesting/trivial", or "not about program comprehension", it was filtered out. This resulted in a first aggregated version of the observations with 134 results. In the next step we aggregated the duplicate and similar results and removed the results that were not confirmed in two different sessions. This led to the 18 results summarized in Table V. Finally we reviewed the final results again with the details from the original transcripts, protocols, and the literature and phrased the findings.

2.3.2. Online Survey. We also took several actions to increase the validity and reliability of the survey results. We designed the answer options based on previous findings (see Section 7), the input of fellow developers, and discussions with experts (researchers and experienced developers). From surveying the literature we came up with the first set of questions. We then iterated these questions and answers with three developers from partner companies (not participating in the observational study). Finally, we sent an early draft of the survey to two experts in the field of knowledge management and software engineering and asked for feedback. Moreover, we included additional optional free text answers to allow participants to add missing options if applicable.

Once the first version of the questionnaire was stable, pilot testing was also an important measure to increase the reliability of the survey and get feedback from the respondents whether the questions can be answered easily and clearly. In total, we conducted four iterations to refine the survey, each with three or more participants. After the first iteration, we learned that the questions needed too much time (more than 30 minutes). Therefore, we decided to reduce the number of questions and focus only on the knowledge aspect of comprehension. After the second iteration, we *unified* the question types to increase respondent experience and introduced the frequency *scales* to give respondents more flexibility in giving their opinions. After the third iteration, we refined the look and feel, unified the terminology, and improved questions for which participants needed longer time. We also added explanations and examples and gave respondents the option to contact us if they had questions or feedback. After the last iteration, no significant changes were done to the survey.

When recruiting participants, we used as many and as diverse channels as possible. The list of channels and statistics of respondents are described in Section 2.4. As we expected many of our recruited participants to be German native speakers, we *translated* the survey to German to minimize misunderstandings of terms. We also tested the survey on different browsers and platforms to ensure the usability of the response forms.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Before analyzing the collected data, we checked whether the received responses were complete, *removing incomplete responses* [Tuten et al. 2002]. This ensures the trustworthiness of the data, since people who did not complete the survey might be mainly interested in the incentives and are not committed enough. In total, we received 2,459 responses of which 1,477 were complete.

We ran several analyses, including within-options, within-questions, within-subjects, and between-subjects analyses. Within-options analysis confirms whether the ratio of the respondents having opposing (i.e., positive and negative) opinions to one option (i.e., a possible answer for one question) is significant. For instance, this allows to check whether the ratio of developers who need to know "who has experience with a piece of code" and those who do not is significantly different. Within-questions analysis allows to confirm the rank of the options within one question, e.g., whether the order of the most, second most, third most popular knowledge channels to share knowledge is significant. Within-subjects analysis allows us to confirm preferences of developers. For instance, those who choose A typically choose B as well or the more frequently developers encounter issue A, the less frequently they encounter issue B. Finally, between-subjects analysis allows us to identify and compare common subpopulations of respondents, such as the group of developers who prefer this channel versus the other group who prefers the other channel. These analyses are important because the goal of the survey is to identify trends and dependencies rather than quantify ratios of the whole population of developers (see Section 2.1 and Section 6).

For the analyses we first used descriptive statistics to summarize the answers on knowledge needs, channels, and problems. Since reporting average values of semantic scales (i.e., "never", "seldom", "often", "usually") can be misleading [Rosnow and Rosenthal 2007, Ch. 5 and 10], we calculated and compared the absolute frequencies, relative ratios, and average values of the scales. Therefore, we used a simple numeric mapping of the scales: 1 for "never", 2 for seldom, 3 for "often" and 4 for "usually". This allowed us to run the following statistical tests to check the significance of the results.

For the within-options and within-questions analyses, we summed up the four scales ("never", "seldom", "often", and "usually") to two categories (positive and negative) and conducted pairwise Chi-Square Tests of Independence [Rosnow and Rosenthal 2007, Ch. 15]. We consider "often" and "usually" to be a positive answer (i.e., popular need or popular usage) whereas "seldom" and "never" to be rather negative (i.e., unpopular need or unpopular usage). For the within-subjects and between-subjects analyses we systematically ran three levels of statistical tests. First, we combined all possible pairs of questions and ran a correlation test and a Chi-Square Test of Independence. After having both test results, we filtered the correlations with a ratio greater than 0.3, which we consider as threshold for a rather strong correlation, and analyzed those manually. Then, we cross-tabulated each of the questions with a strong correlation ratio to determine which subpopulation made the difference, using the standardized residual analysis. Finally, we ran the Mann–Whitney–Wilcoxon test (also know as Wilcox rank sum test or simply Wilcox test) on the original scale to confirm the trends on a binary scale (positive and negative). Wilcox test is considered appropriate for ordered semantic scales and subsamples of different sizes, as in our case [Winter and Dodou 2010]. We report on statistically significant relationships supported by *statistical tests* to exclude that the results occurred by chance [Anderson and Kanuka 2003]. We also share the data, forms, test scripts, and test results on the Internet *www.teamweaver.org/KOMA_Survey/* and encourage replication of the study and new analysis of the data.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Table IV. Overview of Participants in the Observational Study

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	Understanding other's code
P2	C2 (DE)	3	Technical documenter	Fleet management		PL/ SQL, Oracle SQL Developer	0.25	Documenting other's code
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	Understanding other's code
P9	C5 (DE)	9	Developer	Automotive software	X	C, NotePad ++	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	Reviewing other's code
P12	C5 (DE)	8	Developer	Automotive software	X	C, Eclipse, SourceInsight	8	Reviewing other's code
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	Reviewing other's code
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

2.4. Participant Recruitment

2.4.1. Observational Study. Participants had to work for a software development company and spend most of their time coding. We excluded other people, especially students and university researchers, from the study because we wanted 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.

Overall, we conducted 28 observation sessions with developers from seven companies between February and September 2011. Table IV gives an overview of the 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 participants were familiar with the programs they were trying to comprehend during the observation. *Tec. exp.* represents the experience with the technology used in years.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

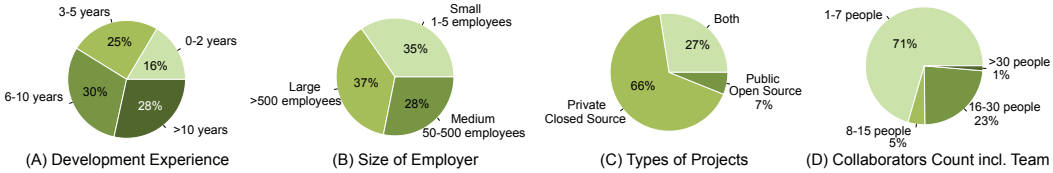

Fig. 2. Demographics of respondents in the online survey, N=1,477.

2.4.2. Online Survey. For the survey, we addressed diverse audiences, including commercial and open-source developers as well as different technical and cultural backgrounds. Our respondent recruiting strategy was twofold. First, we advertised our survey in developer communities such as news sites, mailing lists, and industry conferences. Second, we asked companies to distribute our survey among their developers. To motivate the participation, we raffled two iPods among respondents who completed all questions. Moreover, participants were able to anonymously request a summary of the results and send feedback to us about the study.

Overall, we approached 35 organizations (11 communities and 24 companies) in 2009. We did not receive any response from four communities. Through the remaining 31 organizations, we received a total of 2,459 responses, of which 1,477 were complete (i.e., 60% of all responses). In total, 15 organizations had German backgrounds, eight were international communities, two companies were based in the US, and the rest in Belgium, France, Greece, Hungary, India, Italy, and Serbia. By analyzing the variance between the different organizations (pairwise proportions tests), we were unable to identify any significant difference between the results. The list of participating organizations and the raw answers are also available for download on *teamweaver.org*. To protect respondents' privacy the answers are not associated to their organizations.

Most respondents (86%) had at least three years of development experience and more than half had at least six years of development experience. The respondents were almost equally distributed amongst small, medium-sized, and large companies (Figure 2-B). Two thirds primarily worked in closed-source projects, 7% worked in open-source projects, and 27% worked in both (Figure 2-C).

We also asked about collaboration in development projects. Seventy one percent of respondents collaborated with 1-7 people at least once a week, 5% collaborated with 8-15 people, and 23% with 16-30 people. Only 1% of the respondents had more than 30 collaborators per week (Figure 2-D). Concerning the collaborators' geographic distribution, about 50% of the respondents collaborated only with people collocated in the same room or building, while 20% collaborated only with people located in other buildings, cities, or countries. The rest had a mixture of collocated and distributed collaborators.

3. QUALITATIVE FINDINGS

We summarize the results of the observational study, derive findings, and relate them to the findings of similar studies. Table V gives an overview of our observations and the number of participants and companies that support each of them.

3.1. Comprehension Strategies

By comprehension strategy, we mean the overall approach and activities performed to reach certain comprehension goals, for instance, debugging to determine why a null-pointer exception occurs or asking colleagues to acquire certain knowledge. We observed the following strategies.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Table V. Results of the Observational Study

Observation	# Participants	# Companies
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	11	5
(S7) Establish and test hypotheses	9	5
(S8) Take notes to reflect mental model and record knowledge	9	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
Knowledge Needs and Channels		
(K1) Source code is more trusted than documentation	21	6
(K2) Communication is preferred over documentation	17	5
(K3) Standards facilitate comprehension	12	6
(K4) Cryptic, meaningless names hamper comprehension	10	6
(K5) Rationale and intended usage is important, but rare information	10	5
(K6) Real usage scenarios are useful but rare	5	4

[3]The original version of this paper [Roehm et al. 2012] included a mistake, which is corrected in this version.

(S1) Employ a recurring, structured comprehension strategy depending on context. We noticed that most participants approached tasks using a recurring, structured strategy. Twenty-six 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).

Finding 1 Developers usually follow a recurring, structured comprehension strategy depending on the context, which includes 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 strategies” and what Littman et al. describe as "systematic and opportunistic strategies" [Littman et al. 1987], which 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

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

or opportunistic. The strategies we observed were recurring and depend on the context factors mentioned in Finding 1.

(S2) Follow a problem-solution-test work pattern. In 18 sessions, we observed that participants followed a generic 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, for instance, which code has to be changed and which code has to be added.

For example, P3 tried to fix a coloring bug in the GUI. 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 an 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 an 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 [1976] observed that modifying software generally involves three phases: understanding the existing software, modifying the existing software, and revalidating the modified software. Maalej and Happel [2010] also found that about 25% of developers describe their work by using problem-solution phases.

Finding 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. Twenty-one participants worked on applications that exhibit a user interface and 17 of them used the GUI 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 passed only 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 GUI form and verifying that these values are stored in the database.

Finding 3 Developers interact with the application user interface to test whether 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 executed the application to understand the source code. Six-

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

teen 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. 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. [2006] who found that, in general, debuggers are frequently used by developers for various tasks.

Finding 4 To comprehend programs, developers need to acquire runtime information. For this developers frequently execute the application using a debugger.

(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 whether it is used otherwise") and not being able to test all possible usages after a modification to check the modification correctness ("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.

Most participants claimed that they aim at saving effort by cloning code and only few reported that they aim at avoiding comprehension. Six participants reported cloning code to reuse existing implementation and save effort compared to writing code from scratch. This finding matches with the results of Singer et al. [1997] and of Kim et al. [2004], who reported that developers do not comprehend system aspects that go beyond the needs of their current tasks. Moreover, several recent studies have reported on developers pragmatic procedure of developers and their propensity to copy-and-paste example code, including the work of Holmes and Walker [2013] on pragmatic reuse and the work of Brandt et al. [2009, 2010] on example-centric programming. Rahman et al. [2012] recently studied the relationship between cloning and defect proneness. They found that clones may be less defect prone than non-cloned code.

Finding 5 Developers try to avoid comprehension by cloning pieces of code if they cannot comprehend all possible consequences of changes.
Finding 6 Developers prefer a unified documentation structure to simplify finding information needed for comprehension.
Finding 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 eleven 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 ap-

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

plication architecture as crucial information [Murphy-Hill et al. 2012b; Vakilian et al. 2012].

We observed that experience also influenced the recognition of data structures and helped in the location of concepts [Adelson and Soloway 1985]. 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 their experience. Participants P2, P9 and P10 ignored parts of the source code, arguing, for instance, that "this part implements calculation, which is not important for my current task". This observation matches with results from Sillito et al. [2005] who found that developers attempt to minimize the amount of code to read.

Finding 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.

Similarly, LaToza et al. [2007] investigated the role of experience in helping developers comprehend programs more efficiently. They found that experts did not visit some methods novices wasted time understanding and understood code in terms of abstractions while novices more in term of code statements.

(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 such as "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 such as "I assume this method fetches the maximum value" (P2), "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 Letovsky and Soloway [1986],Brooks [1983], Mayrhauser et al. [1998], and Ko and Myers [2004]. According to these authors, questions lead to informal hypotheses that are verified by developers.

Finding 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 transient memory during comprehension activities. These notes varied from single function names, mappings between identifiers 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 with information on control or data flow or both. 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.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

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

3.2. Tool Usage

We made four main observations about tool usage, that is, 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. Two participants from company C5 used SOURCEINSIGHT[3] to view source code but we did not observe them utilizing the built-in program comprehension features. P3 reported to use WIRESHARK[4], a network packet analyzer, to inspect network communication, while P4 reported to use SELENIUM[5], a macro recorder of user interactions, to simulate user interactions for testing purposes.

Finding 11 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. [2006], who reported on the use of standalone tools in addition to an IDE in a bug fixing scenario. Singer et al. [1997] and Maalej [2009] also found that developers complain about loose integration of tools, which might hinder comprehension as information required is scattered across different tools.

Finding 12 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. He changed the name of that constant in its definition and examined the locations of the resulting compiler errors. P4 used the compiler to find out where he inconsistently adapted a copied piece of code, for instance, inconsistent changes of variable names. P7 used the compiler extensively to check which methods in a copied code block are necessary as described in observation S5. P15 and P18 used the compiler messages to find error locations. Starting from the message "variable x undefined" they performed a full-text search for the code locations of variable x.

Finding 13 The compiler is used by some developers to elicit structural information such as dependencies and usage locations of code elements.

---

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

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

(T4) Tool features for comprehension are unknown. We made an unexpected observation with P3. In order to find all code locations where a specific constant is used, P3 changed the name of the constant in its definition and inspected resulting compilation errors. 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 eight years of professional experience and six years using ECLIPSE. P22 solved the same problem by performing a full-text search for a method name. These observations match with results of Sillito et al. [2005], who reported that developers used tools inefficiently and with the work of Murphy-Hill and Murphy [2011] and Murphy-Hill et al. [2012a] who reported that developers are unaware of common commands and features in the IDE.

Finding 14 Developers do not know some standard features of tools.

3.3. Knowledge Usage

We made six main observations about knowledge usage during program comprehension, in particular on how knowledge was documented and shared and which knowledge was missing to developers.

(K1) Source code is more trusted than documentation. Twenty-one participants reported that they get their main knowledge 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. For that, people try to avoid documentation."

This result supports previous work, who reported that documentation is seldom kept up-to-date [Forward and Lethbridge 2002; LaToza et al. 2006; Lethbridge et al. 2003]. Singer et al. [1997] also reported that source code is read frequently while documentation is not.

Finding 15 Source code is a more trusted source of information than written documentation, mainly because documentation is often non-existent or outdated.

(K2) Communication is preferred over documentation. Seventeen participants reported that communication with colleagues is a more often used channel to access knowledge 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 knowledge needed in the documentation".

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

This result is consistent to the findings reported by LaToza et al. [2006], who emphasized the importance of people to gain knowledge compared to written documents.

Finding 16 Communication with colleagues is a more important channel to access knowledge 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 knowledge seekers whereas the advantage of written documentation is that it is reusable.

(K3) 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 the 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. Three participants from C5 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 [2002], who reported that "regularities in the design, and especially in the naming of functions and data, may greatly facilitate concept location".

Finding 17 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.

(K4) 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[Soloway and Ehrlich 1984]. 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 five 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. Three 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".

Finding 18 Cryptic, meaningless names hamper understanding of a piece of code.
Finding 19 Naming conventions can help to mitigate this effect but if they are too complicated they can have a negative effect.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

(K5) Rationale and intended usage is important but rare information. We observed ten participants who 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 confirmed 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. [2006] that understanding the rationale behind code is a big problem for developers.

Finding 20 Knowledge about implementation rationale and intended ways of using a piece of code helps to comprehend code but this information is rarely documented.
Finding 21 There is a gap between the interest of developers in this information and the lack of documenting it for their own code.

(K6) 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" is 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 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.

Finding 22 The way in which end users use an application is a helpful context information in program comprehension.
Finding 23 In many cases this information is missing.

4. QUANTITATIVE FINDINGS

We summarize the results of the online survey along the three studied aspects: knowledge needs, knowledge exchange channels, and problems encountered. We quantify the popularity of the studied aspects and identify *relationships* between them.

4.1. Knowledge Needs

We asked developers to indicate how frequently they encounter problems due to a missing piece of knowledge needed in a particular development situation. We distinguished between four situations: fixing a bug, reusing components, understanding others' code, and implementing a feature. The studied knowledge needs (i.e., missing pieces of knowledge) and the results are summarized in Figure 3. The modes indicate the most frequently selected value by the respondents. The within-options analysis showed that the differences in the frequency ratios are statistically significant ($p < 0.04$ for pairwise Chi Squared test of independence) except for the needs marked with **. For those needs there is no significant difference between the ratio of developers that confirm a frequent encounter and those who do not. The within-questions analysis confirmed that the showed order of the needs is statistically significant ($p < 0.03$ for pairwise Wilcox tests) except for the needs highlighted with the binding vertical lines.

The survey respondents confirmed several findings from the observational study. More than 83% of the respondents claimed that they often or usually encounter problems due to lack of information about "what is the program supposed to do" or "whether a component provides a certain functionality". This shows the importance of *functionality* knowledge for comprehension, supporting Finding 3 and explains our observations of developers taking the role of users and looking for usage information during

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Problems encountered due to missing knowledge	Frequency: Never (rarely) / Seldom (monthly) / Often (weekly) / Usually (daily)	Often - Usually Count (%)	Mode
Fixing a bug		**(70,1%)**	**Usually**
How to reproduce the bug		1333 (93,7%)	Usually
What is the cause of the bug		1267 (89,7%)	Usually
What should be done to fix the bug		1154 (83,0%)	Usually
Did someone else fix similar bugs		747 (53,4%)	Often
Reusing a component		**(69,8%)**	**Often**
Whether the component provides certain functionality		1153 (83,8%)	Usually
How to use the API (i.e. create and use objects)		1135 (82,4%)	Usually
How to configure the component		1035 (75,3%)	Often
How to extend existing functionality		791 (57,6%)	Often
Who has experience with the component **		681 (49,7%)	Seldom
Understanding other's code (e.g. for review or documentation)		**(59,6%)**	**Often**
What is the program supposed to do		1190 (85,0%)	Usually
What was the developer's intention when writing this code		1025 (73,5%)	Often
Why was this code implemented this way		733 (52,4%)	Seldom
Who has experience with this code **		677 (48,5%)	Seldom
Who wrote this piece of code		538 (38,5%)	Seldom
Implementing a feature		**(59,0%)**	**Often**
What data structures and algorithms can I use		948 (68,7%)	Usually
How can I use a particular feature in my development tool		862 (63,0%)	Often
Am I following all development conventions		839 (61,1%)	Usually
How did other developers implement similar functionality		829 (59,4%)	Often
How are other developers going to use my code **		717 (51,8%)	Seldom
Which design patterns could be applied here **		667 (49,6%)	Often

Fig. 3. Frequencies of knowledge needs encountered in the different development situations. ** indicates that the difference between ratios of positive and negative answers is not statistically significant. The vertical bars binding two rows (needs) indicate that there is no statistically significant difference in the rank of these needs.

program comprehension (Findings 22 and 23). In addition respondents also confirmed the importance of *rationale* knowledge for software comprehension (Finding 20). More than half of respondents agreed that they encounter problems at least weekly due to missing knowledge about "why was this code implemented this way", "what was the developer's intention when writing this code". Respondents also confirmed our observations that many IDE features are unknown, and that standalone tools with redundant functionality are used instead (Findings 14 and 12). 63% of respondents rated problems due to missing information on "how to use a particular feature in the development tool" to occur often (i.e. at least weekly).

Overall, respondents rated the knowledge-missing problems during bug fixing tasks as most frequently encountered (mode: usually). The need to know "how to reproduce the error" ranked number one for the whole survey, followed by "what is the cause of an error", "what is the program supposed to do", and "whether the component provides certain functionality". Remarkably, 85% of the respondents usually or often encountered problems related to these needs. None of the needs of the situation "implementing a feature" were rated as high. In each situation, at least one need is encountered often or usually by more than 2/3 of the respondents, which confirms the relevance of the studied needs and the findings from previous studies [Ko et al. 2007; Sillito et al. 2008; Fritz and Murphy 2010].

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Channels to Access Knowledge	Consulted… Never Seldom Often Usually	Count Often - Usually	Mode
Other People		1170 (81,6%)	Often
Internet		**2993 (70.1%)**	**Usually**
Web search engines (e.g. Google, Yahoo)		1170 (81,4%)	Usually
Public documentation / web pages I know		1081 (76,0%)	Usually
Forums and mailing lists **		742 (52,7%)	Often
Project Artifacts		**3486 (61.8%)**	**Often**
API description		1076 (75,8%)	Usually
Comments in source code		990 (69,3%)	Often
Issue and bug reports		895 (63,1%)	Often
Commit messages		525 (38,1%)	Seldom
Personal Artifacts		**1658 (39.1%)**	**Seldom**
Work item/task descriptions (To-dos) **		710 (50,3%)	Often
Personal e-mails		906 (42,8%)	Seldom
Personal notes (logbooks, diaries, post-it's)		339 (24,1%)	Never
Knowledge Management Systems		**1150 (28.0%)**	**Never**
Project or organization Wiki		532 (38,5%)	Never
Intranet		437 (31,8%)	Never
Experience Databases / groupware systems		181 (13,4%)	Never

Fig. 4. Channels used by developers to access knowledge about software. ** indicates that the difference between ratios of positive and negative answers is not statistically significant.

Based on our qualitative findings (Findings 16 and 8), we expected that information about people, including "who has experience with the code", "who wrote this code", "who has experience with a component", or "who solved similar errors" to be very popular among respondents. Surprisingly, this information was the least sought by respondents. About half of the respondents rated these knowledge-need problems to occur seldom or never, resulting in a mismatch between the observed and the perceived importance of *who-knows-what* knowledge. By cross tabulating the responses, we found significant relationships between knowledge needs about people on the one hand, and the number of collaborators and company size on the other hand ($p < 1e-05$ for a Chi Square Residual Analysis and $p < 0.019$ for a pairwise Wilcox test). Respondents working in small teams (interacting with up to seven people) need to know "who has experience with the code they are understanding or the component they are reusing" *less frequently* than those working in larger teams (interacting with 8-16 people). Most respondents who never need to know "who has experience with the component being reused" work for companies with less than 50 people. In contrast, developers who usually need to know this work for large companies with more than 500 people. This reflects the finding of Grubb and Begel [2012] that larger organizations find "who" questions very important. Mockus and Herbsleb [2002] discussed factors influencing difficulties in locating "who-knows-what" knowledge in software organizations.

Finding 24 Developers heavily rely on communication and personal experiences for program comprehension tasks. However, only developers working for large organizations encounter problems in knowing who has this experience.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

4.2. Knowledge Channels and Tools

In two further questions, we asked developers about the channels and tools they use to access knowledge (i.e., supply their knowledge needs) and to share their experiences with others. We grouped the channels into the following categories: people, Internet, project artifacts, personal artifacts, and knowledge management systems. We first summarize the use of the channels for knowledge access (Figure 4) and then for knowledge sharing (Figure 5). The within-options analyses showed that the differences in the frequency ratios are statistically significant ($p < 0.02$ for the pairwise Chi Squared test of independence) except for the options marked with **. The within-question analysis confirmed that the showed order of the options is statistically significant ($p < 0.02$ for pairwise Wilcox tests).

Confirming our qualitative Finding 16 and in accord with previous studies [Ko et al. 2007; Herbsleb and Mockus 2003], personal channels were the most popular to access knowledge. Eighty one percent of the respondents consult other people often or usually to supply their knowledge needs.

Web search engines such as Google and Yahoo were as popular as people. On average, the category Internet ranked first based on mode and second based on the ratio of often-usually. Based on modes, project artifacts ranked third (often), personal artifacts fourth (seldom). About 40% of respondents use personal comments in action items and log books to access knowledge about the software, confirming our qualitative findings about the importance of personal notes for externalizing knowledge about programs (Finding 10).

Respondents also supported Finding 11 about the rare usage of research tools by developers. We found that the usage of knowledge management systems by respondents is very rare. Less than one third use Intranets, experience databases, or groupware systems on a regular basis. Even Wikis remain unpopular with "never" as mode.

Analyzing the relationships between participants' demographics and their preferences, we identified a consistent and significant relationship between the employer's size and the frequency of using Internet channels, as shown in Table VI-A. Developers working in small organizations use Internet channels such as forums and search engines most frequently to access knowledge ($p < 0.05$ for pairwise Wilcox tests), while developers working in large organizations (>500 employees) use those channel least frequently ($p < 0.01$ for pairwise Wilcox tests).

A similar trend exists between respondents' open-source experience and the frequency of using Internet channels (Table VI-B). Developers who have open-source experience use significantly more web forums, web search, and web documentation than developers without such experience ($p < 0.0003$ for pairwise Wilcox test). Developers who work only in open-source projects use these channels most[6].

Finding 25 Developers' open-source experience and the size of their employers influence their knowledge sharing behavior.

Concerning knowledge sharing, the channels studied are overall less frequently used (Figure 5). On average, only 41% of respondents use these channels usually or often for sharing knowledge (compared to 56% for accessing knowledge). We found two possible interpretations for this. Developers either access knowledge more frequently than sharing it, or the studied channels better support developers' knowledge access than their knowledge sharing behaviors.

Looking at the single channels, we found similarities as well as differences between using them to access and share knowledge. There is an agreement between knowledge

[6]Note that there is a difference in the ratios between only open-source developers and open-source and closed-source developers. However this difference is not statistically significant.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Channels to Share Knowledge	Used… Never Seldom Often Usually	Count Often - Usually	Mode
Personal communications or emails		1178 (82,1%)	Usually
Project Artifacts		**3115 (56,2%)**	**Often**
Comments in source code		974 (69,8%)	Often
Shared documents		739 (53,0%)	Often
Comments of a bug report **		709 (51,0%)	Often
Comments in commit message **		693 (50,9%)	Often
Personal Artifacts		**340 (24,9%)**	**Never**
Personal notes (logbooks, diaries, post-it's)		340 (24,9%)	Never
Knowledge Management Systems		**871 (21,2%)**	**Never**
Internal Wikis or groupware systems		469 (34,11%)	Never
Internal mailing lists and forums		267 (19,4%)	Never
Experience Databases / groupware system		135 (9,9%)	Never
Internet		**285 (20,7%)**	**Never**
Public mailing lists or forums		285 (20,7%)	Never

Fig. 5. Channels used by developers to share their knowledge about the software. ** indicates that the difference between ratios of positive and negative answers is not statistically significant. The vertical bars binding two channels indicate that there is no statistically significant difference in the rank of these channels.

consumers and providers about the popularity of personal communication (see Finding 16). Again about 82% of respondents usually or often use personal channels to share their experience with others. Informal comments in source code are similarly popular among knowledge consumers and providers, with 70% ratio of usually or often for both (Finding 10).

Respondents also confirmed that knowledge management systems are an unpopular means to share and access knowledge about software in their projects (mode is never for both) [Desouza 2003]. This result supports our qualitative finding that practitioners barely use dedicated state-of-the-art tools developed by the research community for comprehending programs and accessing or sharing knowledge about it.

Finding 26 The informal channels such as personal communication and comments are more popular to share knowledge about programs than the formal channels such as knowledge management tools and mailing lists.

By cross-tabulating the results, we also found a consistent trend amongst consumers and providers concerning the usage of emails. The higher the distance between collaborators, the more frequently emails are used to access and share knowledge. While only 38% of developers who collaborate with people in the same room often use emails to share knowledge, this rate increases to 41% with collaborators in the same building, 46% in the same city, 47% in the same country, and 55% in different countries. While the differences between two consecutive distribution levels (e.g., same room and same building or same city and same country) are not statistically significant, this difference between developers working in the same room, same building, or same city one the one hand and same country or different countries on the other hand is statistically significant ($p < 0.009$ for pairwise Wilcox tests). The same observation holds between the

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Table VI. Respondents Demographics and Popularity of Internet Channels to Access Knowledge (Often or Usually)

Internet channels	(A) Organization size		
	Large (>500 empl.)	Medium (50-500 empl.)	Small (<50 empl.)
Public Forums	43%	55%	61%
Search Engines	76%	83%	87%
Pub. Documentation	70%	77%	82%
Internet channels	**(B) Project type (open source experience)**		
	Closed source only	Closed & open source	Only open source
Public Forms	47%	63%	65%
Search Engines	78%	87%	87%
Pub.Documentation	73%	81%	86%

distribution of the collaborators and the use of internal mailing lists to share knowledge. In this case the rates go from 18% for working with collaborators in the same room to 30% with collaborators in different countries ($p < 0.01$). We also evaluated whether there is a correlation between preferring personal communication to access and sharing knowledge and the company size. The results do not show a significant relationship.

When conducting a within-subjects analysis of the knowledge access and knowledge sharing behavior we identified several interesting trends. The more developers consult comments in source code when trying to answers their questions, the more they also use this type of comments to share information. This trend is also statistically significant for personal notes, wikis, experience databases, and commit messages ($p < 1.74e - 91$ for pairwise Wilcox tests). A similarly strong relationship exists between the web channels. The more developers use a web-search engine to find answers to their questions, the more they also use public documentation and forums for the same reason.

Conducting a between-subjects analysis, we found a disagreement between knowledge consumers and providers about the popularity of informal artifacts such as commit messages and public forums. While 51% of respondents frequently share knowledge in commit messages, only 38% use them to access knowledge ($p < 1.557e - 10$ for pairwise Wilcox tests). The strongest disagreement is about public mailing lists and forums. While 53% of respondents regularly seek for answers of their questions in these channels, only 21% regularly share their experience through them ($p < 2.2e - 16$ for pairwise Wilcox test).

Finding 27 While some channels are preferred to access knowledge by developers, others are rather preferred to share knowledge.

Communication structure in projects, project policies, and tool support might raise this mismatch. Senior developers, developers of widely used components, or developers explicitly responsible for sharing knowledge might do a disproportionate amount of the sharing [Cataldo and Herbsleb 2008]. Restrictive closed-source projects might encourage developers to access public knowledge but discourage them to share their experience to the public. Moreover, organizations' "commit policies" enforce writing a commit message but not necessarily reading them, for instance, in an "update policy" or searching and browsing them. Finally, state-of-the-art tools lack advanced features to search for such informal information or to easily access it from different tools.

4.3. Problems with Accessing and Sharing Knowledge

We asked developers to rate problems encountered during knowledge access and knowledge sharing. The list of problems and the results are depicted in Figure 6. The within-options analysis showed that the differences in the frequency ratios are statis-

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

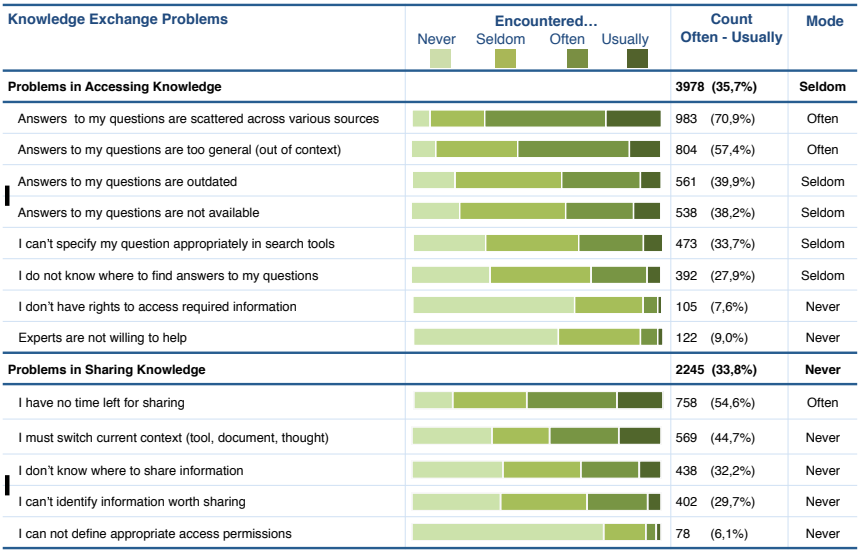

Fig. 6. Problems encountered while accessing and sharing knowledge about software. The vertical bar binding two rows (problems) indicates that there is no statistically significant difference in the rank of these problems.

tically significant ($p < 0.0002$ for pairwise Chi Squared test of independence) for all problems. The within-questions analysis confirmed that the order of the problems is statistically significant ($p < 7e - 09$ for pairwise Wilcox tests) except for the questions highlighted with the binding vertical lines.

The results confirm several qualitative findings. Regarding knowledge access, respondents rated the *information scatter across various tools* as the most frequently encountered problem. Only 7% of respondents never/rarely encounter this problem. This result confirms the findings of Fritz and Murphy [2010] and Maalej [2009]. This result also presents a possible justification for our observation from the qualitative study, that developers seek for standardization to facilitate comprehension (Finding 6 and Finding 17).

The problems "information accessed is out of context" and "answers to questions are outdated" ranked second and third. At least 40% of respondents encounter one of these problems often or usually (i.e., at least once a week). This supports our Finding 15 about trusting the source code more due to outdated documentation.

Regarding knowledge sharing, respondents indicate the *lack of time* as the most frequent problem (mode: often). Moreover, about 45% of respondents frequently encounter problems due to switching current working contexts, for instance, switching the tool to share information. This is indirectly related to lack of time as well. We think that this result is related to our qualitative finding between the interest of developers in comprehension information and the lack of documenting it in their own code (Finding 21).

Overall, more than 50% of the respondents unexpectedly rated the majority of the problems (10 of 13) to occur seldom or never. In particular, most respondents (> 90%)

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Table VII. Work Experience and Knowledge Sharing Problems

Work experience	0-2 years	3-5 years	6-10 years	>10 years
Problems in identifying **what** to share (often or usually)	38%	32%	30%	22%
Problems in identifying **where** to share (often or usually)	35%	32%	32%	20%

rarely or never encounter access-right problems to access or share knowledge. Similarly, there is rarely a problem with willingness of experts to help by sharing their experiences. This result contradicts to some extent the established theory that for knowledge workers [Gilder 2013; Cabrera and Cabrera 2002] “knowledge is power” and experts are rather reluctant to share their knowledge. This seems to not apply in the software engineering domain.

We also found a relationship between the number of collaborators and the frequency of sharing problems. The more collaborators respondents have, the fewer problems they encounter in identifying which knowledge to share ($p < 0.007$ for Kendall’s rank correlation test). This relationship is an indicator – although not an evidence – for a causal relationship between the collaboration intensity and the easiness of identifying which knowledge to share and to whom [Dagenais and Robillard 2010].

Finding 28 The more people developers know, the easier they can identify which experience is worth sharing, with whom, and how.

A similar relationship exists in the responses between the work experiences and the frequency of encountered problems. We found developers with shorter work experience claim to encounter knowledge sharing problems more frequently, as shown in Table VII ($p = 2e - 05$ for Kendall’s rank correlation test). This trend is similar to our qualitative findings about the role of experience in software comprehension (Finding 8). We found two possible explanations for this observation. Experienced developers either do not admit to encounter these problems, or their development experience also involves knowledge sharing experience. With time, they learn best practices to deal with knowledge sharing problems. They also learn (based on what they have learned from previous projects) to better "judge" which knowledge is worth sharing and where.

Finally, we asked developers whether they would share knowledge if their development environments would automatically capture it. The results are shown in Figure 7. Overall, respondents are rather willing to share knowledge automatically. More than 85% of respondents would share the context of an encountered problem and actions performed to solve it. About 75% agreed to share other knowledge such as the goal of changing a particular piece of code or sources consulted while solving a particular problem. This kind of knowledge would facilitate program comprehension tasks as our qualitative and quantitative results show (Section 3.3 and Section 4.1).

When it comes to more *sensitive* information such as the confidence with a particular topic, actions that led to a problem, or developers’ names, respondents are more cautious [Desouza 2003]. Up to 40% of respondents remain undecided or prefer not to share such information. This can be explained either by the privacy concerns of developers (e.g., possibility for employers to detect incompetence) or by the mistrust of the current tools to automatically derive correct information.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

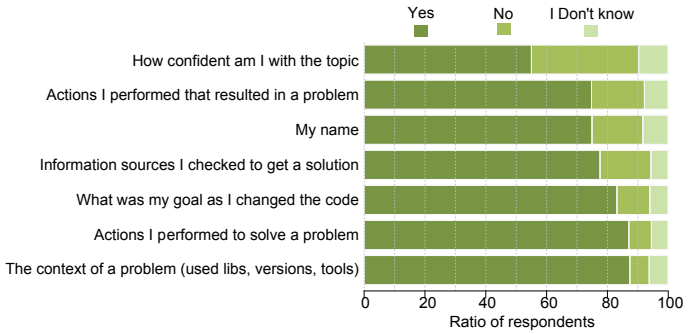

Fig. 7. If your development environment would allow you to automatically share information on your experiences, which information would you share with colleagues?

5. DISCUSSION OF FINDINGS

We summarize the most important findings of our study and discuss their implications for researchers, tool vendors, and practitioners.

5.1. Gap Between Research and Practice

Our results reveal a gap between the state-of-the-art and the state of the practice in program comprehension. On the one hand, our results question the usefulness of program comprehension tools suggested by research. On the other hand, the perception of program comprehension seems to be different between developers and researchers.

We were surprised that none of the participants in our observational study used dedicated program comprehension tools such as visualization, concept location, or software metric tools (see Finding 11). State-of-the-art program comprehension tools were either unknown or rarely utilized even by experienced developers (see Finding 14 and Figure 3). Developers seem to prefer basic tools such as compilers, text editors, and notepads to comprehend programs and capture knowledge about them (see Findings 12, 13, and 10). This raises the question about the impact and usefulness of tool research in program comprehension. Similarly, our survey confirmed the clear trend that dedicated knowledge management tools are barely used to access and share knowledge about programs. Instead, developers seem to prefer integrated and informal solutions such as comments or well-known trusted channels such as personal communication, emails, and web pages.

We were also surprised about the perception of developers to program comprehension and their "openness" about this. While researchers originally focused on "systemizing" program comprehension, developing approaches, models, and tools to understand, visualize, and navigate the source code, our qualitative and quantitative results show that developers in industry rather practice software comprehension in a pragmatic way. First, developers often need to understand the software as a whole including its functionality, usage, documentation, and errors. Second, developers try to avoid comprehension whenever possible (see Finding 5) focusing rather on the expected output of the main development task such as extending a functionality or fixing a bug. In recent years, researchers have begun to realize this point, which has lead to several studies and approaches about a more pragmatic comprehension Brandt et al. [2010]; Holmes and Walker [2013]; Rahman et al. [2012].

There are several possible reasons behind such a gap [Singer 2013]. First, research results and their benefits might be too abstract, complicated, or inapplicable for industry. Second, there might be a lack of knowledge about available tools and their

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

usefulness among practitioners. Third, there might be a lack of trust in new tools. Finally, the high familiarization effort for the new tools might discourage developers from trying them out. Research effort should be focused on how research results can be used to support developers in their everyday work and on how software comprehension features can be integrated into the workflows depending on the task.

The observation that some developers do not know standard features such as the ECLIPSE feature REFERENCES (see Finding 14) emphasizes the need to train developers to use tools efficiently and to proactively inform them about new features [Murphy-Hill and Murphy 2011; Murphy-Hill 2014]. Developers might not know that a specific feature exists, what it is good for, how to use it effectively, or in which situations can the feature help. Overall, we wonder: how many features can be incorporated into a tool such as ECLIPSE before it becomes too overwhelming for a developer to first comprehend its features and then use them appropriately to comprehend the programs.

We think that software engineering researchers and tool vendors should study this gap in detail. If it is confirmed, new agendas and solutions approaches should be proposed. Researchers should investigate the scope of the gap, the reasons behind it, and align their efforts to the needs of industry.

5.2. Context Awareness in Program Comprehension

We studied how program comprehension is performed, which knowledge is needed, and how it is accessed and shared. Overall, the results showed that program comprehension does not follow a "one size fits all" approach. Instead it rather "depends on the concrete context". This includes the task at hand, the project and technological setting, and the backgrounds and habits of the developers themselves.

When studying comprehension strategies, we found that developers "select" the right strategy depending on the type of task, its criticality, technology used, as well as development and project experience (see Finding 1). For example, where to start reading and comprehending the program and how to filter irrelevant parts strongly depends on the previous experience and the type of tasks (see Finding 8). While our results includes a few hints about which factors might influence the strategy and concrete decisions of the developers, only a more differentiated and statistically significant factor analysis can clarify the influence of the context factors.

Context information also seems to play an important role when supplying the knowledge needs of developers in program comprehension scenarios. For instance, the context of the task in which the program was developed, including the rationale, the developer's intention when writing the code, and the intended usage of the program, seems to be frequently needed by developers (see Finding 4, Finding 20, Finding 23 and Figure 3).

Unfortunately, our findings reveal that context information is typically implicit and not captured. For example, respondents rated missing context of documented knowledge about programs as one of the most frequent knowledge access barriers. Similarly, context switching overhead was rated as one of the most frequent knowledge sharing barriers.

One possible strategy to overcome these barriers is an integrated, context-aware tool support for knowledge exchange: enabling a (semi-) automatically capturing and sharing of knowledge with its context by observing developers' interactions [Maalej et al. 2014]. Moreover, insights about the work patterns of developers from this and other studies (e.g., Murphy et al. [2006]; Parnin and Rugaber [2009]) can be used to build personalized 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 [Roehm and Maalej 2012].

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

The implementation of such tools would require the instrumentation of the IDE and continuous observation of the work of developers. To this end, we found that developers are generally willing to share information about their work collected by the IDE automatically. The vast majority of developers agree to share non-personal information such as artifacts used and experience made while comprehending software (see Figure 7).

5.3. User Behavior and Usage Data for Comprehension

Developers showed a similar behavior to a user of a software system at the beginning of a program comprehension activity (see Findings 23 and 21). For instance, they heavily interact with the user interface to simulate and test expected program behavior (see Finding 3). Usage data describing how the system or a particular part of it is actually being used by its users seems to be helpful for developers to comprehend software, not only to reproduce a buggy situation, but also to get familiar with a program and its internal behavior.

Another similar interesting aspect calling for further investigation is the avoidance of a deep understanding of the source code by observed developers (see Finding 5). One potential reason is that software comprehension is a hard and time consuming task and consequently is avoided whenever possible. This indicates that Carroll's minimalist theory [Carroll 1998], which suggests people put in the minimum effort to maximize their outcome, is applicable to several of strategies developers used [Brandt et al. 2009; LaToza et al. 2007]. One implication is that future research should investigate the motivation behind this behavior, the contexts in which such behavior occurs and judge if such behavior should be considered positive or negative. Similarly developers used stand-alone tools such as text editors in addition to integrated development environment (see Finding 12). In general, it seems that developers try to avoid complexity as much as possible.

We think that researchers should consider developers as users and investigate how "user-developers" analyze application behavior, how they relate observations to code, and how this behavior can be supported by tools (which, for instance, collect and process usage data at runtime). For example, it would be easier for developers to ask their questions and to set breakpoints in the graphical user interface, which are then automatically associated with source code by the IDE [Ko and Myers 2008]. This would prevent exasperating searches within the source code.

5.4. The Knowledge Sharing Dilemmas

Our results reveal several interesting trends about the capturing and sharing of knowledge about programs – a major challenge for an efficient software maintenance from the project or the organization perspective.

Personal communication is effective but not always efficient. Both our qualitative and quantitative results confirmed that developers prefer personal communication to access and share knowledge about programs (see Findings 16, 24, and 26). The fact that communication is preferred over documentation might have two drawbacks. First, information might get lost when experts leave the organization. Second, experts frequently get interrupted from their tasks. Therefore, practitioners need to assess this tradeoff when deciding about how to access and share information about program comprehension. *Internet channels* are rated almost as popular as personal communication (see Figure 4 and Figure 5). It would be interesting to observe this trend and compare the popularity of personal versus Internet channels in the future.

Experience is a key but remains tacit. Personal experience plays an important role in program comprehension, for instance, in selecting the comprehension strategy (see e.g., Finding 1) or when externalizing and recording knowledge about programs (see

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

e.g., Finding 10). One would expect that information about people's experience (i.e. who knows what) is an important piece of knowledge in projects. Surprisingly, developers rated information about people's experience (e.g., who wrote a piece of code or who has experience with a part of the program) to be significantly less popular. Researchers should investigate the reason behind this, answering the question: how do developers know who knows what? Possibly, developers do access "who-knows-what" knowledge frequently but they are unaware about this, for instance, because this knowledge is rather informal and not known as such.

Patterns of knowledge sharing. Our results reveal several patterns for knowledge sharing amongst developers. For instance, we found that the number of collaborators, the company size, and the previous open-source experience correlates with how frequently developers need or acquire a certain information and how they share it with others. Further research could help to develop and evaluate approaches and tools for knowledge exchange adapted to the task, company size, number of collaborators, and their location. When developing new tools and methods, tool vendors should consider that knowledge exchange is influenced by several factors such as company size, the number of collaborators, or their distribution.

Documentation quality versus efficiency. We found that information captured in documentation is often outdated, out of context, or even not understandable, which retains developers from trusting it (see Finding 16). One reason is the lack of time to share knowledge (see Figure 6). Moreover, developers rather often rely on quick and informal communication to answer their questions. The dilemma of documentation quality versus the efficiency of knowledge sharing should be studied further, from both perspectives: documentation creators and documentation users. One mean to reduce this dilemma – in addition to using standardized coding style and meaningful names (see Findings 18 and 17) – is to use standardized documentation styles [Maalej and Robillard 2013]. Practitioners should think about what kind of coding and documentation styles they want to implement in their organization and evaluate the quality of documentation as well as the efficiency of knowledge access based on these styles.

Mismatch between access and sharing behavior. Our results reveal a mismatch in the popularity of channel usage between the seekers and providers of knowledge (see Finding 27). For instance, while knowledge seekers tend to look for "trusted" channels such as project documents or known colleagues, knowledge providers prefer informal channels such as notepads, code comments, or commit messages, which allow them to capture their experiences within the task. This mismatch is also reflected in the popularity knowledge-access barriers. We learned that developers often encounter problems accessing knowledge because the information scatters across various sources which means that developers have to switch their working context in order to access the information. Integrated tool support for context-aware knowledge access and sharing could overcome these barriers. It is important to detect the specific needs and their emergence context and to (semi-) automatically collect and share knowledge with the context included. Frequently accessed knowledge channels should be integrated into development environments.

6. LIMITATIONS AND THREATS TO VALIDITY

When designing and conducting this research, we took a series of measures to increase the reliability and validity of its results, which we described in Section 2.3. In this section we summarize the limitations and potential threats which we are aware of.

6.1. Observational Study

There are several limitations to the internal and external validity of our qualitative results from the observational study. As for the *internal validity*, *participants* might

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

have chosen easy tasks to pretend “good performance” during the observation. While it is impossible to eliminate such a risk in a field observation, we think that this risk is not threatening the results. Most participant were either working on tasks defined by managers based on project and daily priorities or selected tasks from the issue trackers based on priority. Moreover, we did not observe any trivial task and participants had difficulties of different levels in all observed task. In general, 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. Our study was pre-announced in the participating companies with all details about the research and researchers to maximize participants’ trust. We also refrained from digitally recording the interaction or the interviews to reduce the fear of participants that this could be used against them.

Similarly, an inherent threat to validity with think-aloud is that this method “requires additional cognitive activity that often leads to reactivity and invalid reports”[Ericsson and Simon 1993]. By interviewing the participants after the observation session, we ensured that any unusual behavior was intentional and participants were able to explain it. Moreover, since we observed the developers in their environments working with their tools and on their projects, we assume that the additional effort needed to describe and explain what they were doing is rather small and does not influence their behavior significantly.

Another potential threat to the internal validity is that the *observers* might had prejudgment and assumptions. They might considered only clues affirming their 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 independently observed by two observers in at least two different sessions. Moreover, we took a series of additional reliability measures (see Section 2.3) such as the peer debriefing and participant checks to reduce this bias. While we think that this bias is negligible for the findings which either confirm previous studies or are supported by the survey, it should be considered when interpreting the other results of our observational study.

There also 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.

The *aggregation of the observation* into the findings automatically leads to the loss of information and certain mixture between observation and interpretation. While every aggregated report on an empirical study cannot completely eliminate this threat, we tried to support our findings with quotes and observations linked to concrete participants. We also sent the findings to participants and other experts from the field for review prior to publication. However, there remains a certain amount of subjectivity in describing the aggregated results, for instance, choosing the heading and phrasing the findings. Therefore these findings should be interpreted carefully.

As per the external validity, a common limitation for most qualitative studies is the low degree of generalizability.

We are aware that in 45 minutes 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

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

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.

Our observational study was designed to have a strong degree of realism, taking different possible settings into consideration 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. Overall the sample size in the observational study is rather small with 28 participants. However, the distribution of participants includes different company sizes (10-300.000 employees), different experiences (from 2-19 years), different application domains (automotive software, security, web development etc.), different programming languages (C, C#, Java, Python, Delphi, Pascal, VB, .NET), different roles, and different countries – representing a wide range of potential participants. This gives us some confidence that the results have a medium degree of generalizability.

Due to the high effort needed to participate in the observational study, it was difficult to recruit a large number of participants. Therefore, the quantitative findings might be skewed toward a small number of companies. In particular C5 and C7 provided more than half of the participants as they were both large software organizations. C5 is an international supplier for automotive companies, which is based in Germany and has more than 350 subsidiaries across the globe. The subsidiary where the observation took place has about 5500 engineers. C7 is also a large international software company that develops enterprise information management systems. The company is based in Canada and has worldwide more than 5000 employees. The office in Germany where the observation took place has around 600 employees.

6.2. Online Survey

As for any other online survey, ours also has several limitations to the internal and external validity of its results. One common threat to the internal validity of surveys is the *selection of the "right" questions and answer options* (i.e., studied variables), since we had to limit the number of variables. We might have influenced the results by this selection. To mitigate this threat, we carefully designed and tested the survey in several iterations based on the results from the previous studies and the feedback of three experienced developers. We also allowed to extend the predefined answers by other missing options. When analyzing the free text options we did not identify any missing option, which was frequently mentioned by respondents. Similarly, the *selection of the semantic scales* and the concrete wording of the answers might have influenced the results as well. Changing the scales, for instance, to measure the importance of the knowledge needs instead of their frequency, might lead to different results. Therefore, we can make claims only about the exact variables studied in the survey.

Another typical bias on online surveys in which respondents are individually recruited is the *volunteer bias*. Respondents volunteering to answer the questions show typical characteristics such as a higher need for social approval, more-unconventional behavior, and a stronger agreement behavior than non-respondents [Rosnow and Rosenthal 2007, p 218]. To mitigate this threat we used several incentives and means to stimulate the participation of typical non-respondents, including giving gifts for participation, explaining to potential subjects why the research is interesting, and emphasizing the scientific importance of their participation. As we cannot completely eliminate this threat, we kept the characteristics in mind when phrasing the questions (e.g., no agreements disagreements questions) and when analyzing the results.

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

Since we did not identify any indicators for typical volunteer behaviors and since we managed to get a large number of respondents with balanced demographics (as shown on Figure 2) we are confident that this threat does not completely bias the results.

It is also important to note that the correlations identified between the studied variables do not necessary imply causal relationships. There might be *third hidden variable*s, which we did not study in the survey. Such causal relationships can only be confirmed by a controlled experimental design including an experimental and a control group.

Our survey was designed to gain quantitative results with an acceptable level of *external validity* by collecting 1,477 complete answers from 31 different organizations worldwide. Our results on the frequencies and ratios of knowledge needs, channels, and problems cannot be generalized to all software developers. Such a generalization would have required knowing the whole population and drawing a representative random sample, which is very difficult if not impossible. However, due to the large uncontrolled sample, we feel confident that these results can be used as a benchmark for quantifying these aspects of knowledge exchange. However, the general trends – which we identified by the within-options, within questions, within-subjects, and between-subjects analyses – are statistically significant and have a higher level of external validity.

7. RELATED WORK

There are numerous researchers who studied program comprehension in the past three decades. A review of the field can be found in [Detienne 2002]. In general we align relevant related work along two categories: strategies and tools developers use to comprehend software and knowledge needs which emerge during software comprehension tasks. In the first category, Von Mayrhauser and Vans [1996]; Mayrhauser et al. [1998] focused on cognitive processes and mental models of developers during maintenance tasks. Their results are complementary to ours, as we focused on the externalized behavior of developers and its rationale. Singer et al. [1997] empirically studied developers' habits and tool usage during software development. LaToza et al. [2006] conducted a similar study focusing on software maintenance. Both studies were hosted by a single company, while our sample includes developers from various companies and domains. Other researchers including DeLine et al. [2005], Sillito et al. [2005], Ko et al. [2006], and Robillard et al. [2004] studied the behavior of developers during maintenance tasks and program comprehension activities. These studies used an experimental setting implying a more focused scope and a small number of participants working on unfamiliar code. When reporting on our findings we discussed which results from these studies were confirmed by our study. Lethbridge et al. [2003] and Forward and Lethbridge [2002] studied how developers use and maintain documentation. We made similar observations but we also identified additional strategies for software comprehension in general.

Concerning tools, Murphy et al. [2006] studied how programmers use the ECLIPSE IDE by analyzing their interaction data collected by instrumenting ECLIPSE. Maalej [2009] studied tool usage and problems developers face regarding tool integration using interviews and online questionnaires. These studies are complementary to our. We explicitly discussed overlaps between results of related studies and our study during the description of our findings. Moreover, we made additional observations. Perhaps the most interesting is the unfamiliarity of developers with state-of-the-art comprehension tools and features, for instance, inside ECLIPSE.

The question "which knowledge developers need during software comprehension" has been studied extensively in recent years. Ko et al. [2007] studied information needs of developers in collocated teams. The authors observed 17 developers at Mi-

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

crosoft, identifying 21 encountered questions such as "What is the program supposed to do" or "What code could have caused this behavior". Similarly, Sillito et al. [2008] observed 25 developers and identified 44 questions specific to software evolution tasks. The authors also identified problems around answering these questions such as "the information scatter across the tools" or "a missing tool support for asking more refined or precise questions". Both studies focused on information needs formulated in form of questions asked and answered during a programming task. The single needs and problems which we asked our survey respondents to rate, were partly inspired from the findings of these studies. We used a different quantitative method (online survey) asking developers about their subjective perceptions of these needs. Moreover, we focused on the sources of information and overall access and sharing channels and problems of these needs. Robillard [2009] studied obstacles that developers at Microsoft face when learning to use APIs of external components. The author found that "missing information" and "out-of-context documentation" are two of the obstacles. Fritz and Murphy interviewed eleven developers at IBM and identified 78 questions developers want to ask, but for which support is lacking [Fritz and Murphy 2010]. The authors also identified channels where the information sought can be accessed, including source code, change sets, teams, work items, and informal comments.

Most recent studies on knowledge exchange in software engineering focus on analyzing how knowledge is accessed by developers [Bjørnson and Dingsøyr 2008]. Few studies considered the role of knowledge providers [Dagenais and Robillard 2010]. In our survey, we focus on both aspects, which allows for an overview of comparing the channels preferred, problems encountered, and mismatch between both perspectives.

From the methodological point of view, most previous studies on knowledge needs and channels are based on qualitative methods [Bjørnson and Dingsøyr 2008], such as observations Ko et al. [2007]; Sillito et al. [2008], interviews Robillard and DeLine [2010], or case studies [Reichling et al. 2007; Milewski et al. 2008]. Our study provides a mixture of qualitative and quantitative methods, allowing not only to generate observations and hypotheses but also to measure the assessments of the developers themselves. Finally, previous studies on knowledge needed are rather based on small samples, observed within one or two organizations. By collecting 1,477 complete answers from 31 different organizations worldwide we aimed at quantitative results with a high level of external validity. This allows us to provide additional, stronger evidence, refine and extend existing findings.

8. CONCLUSION

In this research, we observed 28 and surveyed 1,447 developers from multiple software companies to get insights into the state of the practice in program comprehension. We focused on the strategies followed by developers to comprehend programs, tools they used, knowledge they needed, and how they access and share this knowledge. Some of our findings confirm results of previous studies, others are new and surprising.

Our results confirm the importance of personal communication for accessing and sharing knowledge about software programs. But surprisingly, participants, in particular those working in small teams, claimed to rarely encounter problems in knowing who-knows-what (e.g., who has experience with a piece of code). Knowledge about implementation rationale and the intended usage of a program helps to comprehend code but this knowledge is rarely documented. Our results also confirm the importance of web resources to access knowledge about the software [Hoffmann et al. 2007; Singer et al. 1997] and informal artifacts such as comments in bug reports and commit messages to share this knowledge.

We found that state-of-the-art tools for comprehending programs and accessing and sharing knowledge about it are either unknown or rarely utilized. This reveals a gap

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

between research and practice in program comprehension. IDE features supporting comprehension are not used or not known to developers. Similarly, dedicated tools to capture and access knowledge are barely used. Instead, developers prefer informal, pragmatic, integrated solutions such as code comments, commit messages, and emails. The next step is to investigate the reasons behind this gap, check the findings, and develop new need-driven tools and approaches.

One of our most interesting findings is that developers put themselves in the role of users at the start of a comprehension process and try to collect and comprehend usage data. We observed developers inspecting the behavior visible in user interfaces and comparing it to expected behavior in bug fixing as well as in other implementation tasks. This strategy aims at understanding the program behavior and structure and at getting first hints for further program exploration – an alternative to reading and debugging source code. This suggests studying the potentials of usage data at the deployment time to facilitate comprehension at the development time.

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 rather than a goal by itself.

Finally, we found that there is no one-size-fits-all approach to comprehend programs and exchange knowledge about it. Most observed developers choose from a set of structured comprehension strategies depending on their current work context. Thereby, context constitutes the task at hand, the technology and framework used, previous knowledge about the program, and the developer's general experience. Work context also influenced developers' knowledge exchange behavior, which also depends amongst others on the tasks, the size of the companies, and the number of collaborators. These trends should be further studied and considered when designing and introducing new context-aware, personalized methodologies and tools for program comprehension, including accessing and sharing knowledge about programs.

Acknowledgement

We are very grateful to the participants of the observational study and the survey. We would also like to acknowledge the contribution of Hans-Jörg Happel, Enrique Garcia Perez, and Zijad Kurtanović in designing the survey and analyzing the data. Moreover, our thanks goes to Martin Robillard, the TOSEM reviewers and editors, the ICSE 2012 reviewers, and the anonymous IEEE Software 2011 reviewers for their constructive feedback on early versions of this paper. 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, and EU-IST TEAM project, grant FP6-35111).

References

B. Adelson and E. Soloway. 1985. The Role of Domain Expenence in Software Design. *IEEE Transactions on Software Engineering* 11, 11 (1985), 1351–1360. DOI:http://dx.doi.org/10.1109/TSE.1985.231883

T. Anderson and H. Kanuka. 2003. *e-RESEARCH Methods, Strategies, and Issues*. Pearson Education.

F. O. Bjørnson and T. Dingsøyr. 2008. Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. *Information and Software Technology* 50 (October 2008), 1055–1068. Issue 11. DOI:http://dx.doi.org/10.1016/j.infsof.2008.03.006

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

B.W. Boehm. 1976. Software Engineering. *IEEE Trans. Comput.* C-25, 12 (1976), 1226–1241.

J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. 2010. Example-centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 513–522. DOI:http://dx.doi.org/10.1145/1753326.1753402

J. Brandt, Ph. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1589–1598. DOI:http://dx.doi.org/10.1145/1518701.1518944

S. Breu, R. Premraj, J. Sillito, and Th. Zimmermann. 2010. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work (CSCW '10)*. ACM, New York, NY, USA, 301–310. DOI:http://dx.doi.org/10.1145/1718918.1718973

R. E. Brooks. 1983. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.

A. Cabrera and E. F. Cabrera. 2002. Knowledge-Sharing Dilemmas. *Organization Studies* 23 (2002), 687–710.

J. Carroll. 1998. *Minimalism beyond the Nurnberg Funnel*. MIT Press.

M. Cataldo and J. D. Herbsleb. 2008. Communication Networks in Geographically Distributed Software Development. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work (CSCW '08)*. ACM, New York, NY, USA, 579–588. DOI:http://dx.doi.org/10.1145/1460563.1460654

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

B. Dagenais and M. P. Robillard. 2010. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. 127–136.

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

Kevin C. Desouza. 2003. Barriers to effective use of knowledge management systems in software engineering. *Commun. ACM* 46 (January 2003), 99–101. Issue 1. DOI:http://dx.doi.org/10.1145/602421.602458

Francoise Detienne. 2002. *Software Design – Cognitive Aspect*. Springer.

K. A. Ericsson and H. A. Simon. 1993. *Protocol Analysis: Verbal Reports as Data*. Massachusetts Institute of Technology.

R. K. Fjeldstad and W. T. Hamlen. 1979. Application Program Maintenance Study: Report to our Respondents. In *GUIDE 48*. Philadelphia, PA.

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

Th. Fritz and G. C. Murphy. 2010. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 175–184. DOI:http://dx.doi.org/10.1145/1806799.1806828

G. Gilder. 2013. *Knowledge and Power: The Information Theory of Capitalism and How it is Revolutionizing our World*. Regnery Publishing.

A. M. Grubb and A. Begel. 2012. On the perceived interdependence and information

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

sharing inhibitions of enterprise software engineers. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work (CSCW '12)*. ACM, New York, NY, USA, 1337–1346. DOI:http://dx.doi.org/10.1145/2145204.2145403

J. D. Herbsleb and A. Mockus. 2003. An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering* 29, 6 (June 2003), 481–494.

R. Hoffmann, J. Fogarty, and D. S. Weld. 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07)*. ACM, New York, NY, USA, 13–22. DOI:http://dx.doi.org/10.1145/1294211.1294216

R. Holmes and R. J. Walker. 2013. Systematizing Pragmatic Software Reuse. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 20 (Feb. 2013), 44 pages. DOI:http://dx.doi.org/10.1145/2377656.2377657

Miryung Kim, L. Bergman, T. Lau, and D. Notkin. 2004. An ethnographic study of copy and paste programming practices in OOPL. In *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*. 83–92. DOI:http://dx.doi.org/10.1109/ISESE.2004.1334896

A. J. Ko, R. DeLine, and G. Venolia. 2007. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th International Conference on Software Engineering, ICSE'07*. IEEE Computer Society, 344–353. DOI:http://dx.doi.org/10.1109/ICSE.2007.45

A. J. Ko and B. A. Myers. 2004. 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*. ACM, 151–158.

A. J. Ko and B. A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 301–310. DOI:http://dx.doi.org/10.1145/1368088.1368130

A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. DOI:http://dx.doi.org/10.1109/TSE.2006.116

Th. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. 2007. Program Comprehension As Fact Finding. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 361–370. DOI:http://dx.doi.org/10.1145/1287624.1287675

Th. D. LaToza, G. Venolia, and R. DeLine. 2006. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*. ACM, New York, NY, USA, 492–501. DOI:http://dx.doi.org/10.1145/1134285.1134355

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

S. Letovsky and E. Soloway. 1986. Delocalized Plans and Program Comprehension. *IEEE Softw.* 3 (May 1986), 41–49. Issue 3.

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

W. Maalej. 2009. Task-First or Context-First? Tool Integration Revisited. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE'09*. IEEE Computer Society, 344–355. DOI:http://dx.doi.org/10.1109/ASE.2009.36

Walid Maalej, Thomas Fritz, and Romain Robbes. 2014. Collecting and Process-

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

ing Interaction Data for Recommendation Systems. In *Recommendation Systems in Software Engineering*, Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann (Eds.). Springer Berlin Heidelberg, 173–197. DOI:http://dx.doi.org/10.1007/978-3-642-45135-5_7

W. Maalej and H.-J. Happel. 2010. Can Development Work Describe Itself?. In *7th IEEE International Working Conference on Mining Software Repositories, MSR '10*. IEEE, 191–200.

Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering* 39, 9 (September 2013), 1264–1282. DOI:http://dx.doi.org/10.1109/TSE.2013.12

V. P. Mark Ackerman and A. E. S. Volker Wulf. 2003. *Sharing Expertise Beyond Knowledge Management*. The MIT Press.

A. V. Mayrhauser, A. M. Vans, and S. Lang. 1998. Program Comprehension and Enhancement of Software. In *Proceedings of the IFIP World Computing Congress - Information Technology and Knowledge Engineering*.

A. E. Milewski, M. Tremaine, R. Egan, S. Zhang, F. Kobler, and P. O'Sullivan. 2008. Guidelines for Effective Bridging in Global Software Engineering. In *Proceedings of the 2008 IEEE International Conference on Global Software Engineering*. IEEE Computer Society, Washington, DC, USA, 23–32. DOI:http://dx.doi.org/10.1109/ICGSE.2008.16

A. Mockus and J. D. Herbsleb. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 503–512. DOI:http://dx.doi.org/10.1145/581339.581401

G. C. Murphy, M. Kersten, and L. Findlater. 2006. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software* 23, 4 (2006), 76–83. DOI:http://dx.doi.org/10.1109/MS.2006.105

E. Murphy-Hill. 2014. The Future of Social Learning in Software Engineering. *Computer* 47, 1 (Jan 2014), 48–54. DOI:http://dx.doi.org/10.1109/MC.2013.406

E. Murphy-Hill, R. Jiresal, and G. C. Murphy. 2012a. Improving Software Developers' Fluency by Recommending Development Environment Commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 42, 11 pages. DOI:http://dx.doi.org/10.1145/2393596.2393645

E. Murphy-Hill and G. C. Murphy. 2011. Peer Interaction Effectively, Yet Infrequently, Enables Programmers to Discover New Tools. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work (CSCW '11)*. ACM, New York, NY, USA, 405–414. DOI:http://dx.doi.org/10.1145/1958824.1958888

E. Murphy-Hill, C. Parnin, and A. P. Black. 2012b. How We Refactor, and How We Know It. *Software Engineering, IEEE Transactions on* 38, 1 (2012), 5–18. DOI:http://dx.doi.org/10.1109/TSE.2011.41

Ch. Parnin and S. Rugaber. 2009. Resumption strategies for interrupted programming tasks. In *the 17th. ICPC*. IEEE.

F. Rahman, Ch. Bird, and P. Devanbu. 2012. Clones: What is That Smell? *Empirical Softw. Engg.* 17, 4-5 (Aug. 2012), 503–530. DOI:http://dx.doi.org/10.1007/s10664-011-9195-3

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

T. Reichling, M. Veith, and V. Wulf. 2007. Expert Recommender: Designing for a Network Organization. *Comput. Supported Coop. Work* 16 (October 2007), 431–465. Issue 4-5. DOI:http://dx.doi.org/10.1007/s10606-007-9055-2

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014

M. Robillard, W. Coelho, and G. Murphy. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Transactions on Software Engineering* 30, 12 (2004), 889–903.

M. Robillard and R. DeLine. 2010. A field study of API learning obstacles. *Empirical Software Engineering* (14 Dec. 2010), 1–30. DOI:http://dx.doi.org/10.1007/s10664-010-9150-8

Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Softw.* 26 (November 2009), 27–34. Issue 6. DOI:http://dx.doi.org/10.1109/MS.2009.193

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

T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. 2012. How do professional developers comprehend software?. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 255–265. http://dl.acm.org/citation.cfm?id=2337223.2337254

R. L. Rosnow and R. Rosenthal. 2007. *Beginning Behavioral Research: A Conceptual Primer* (6th. ed.). Pearson.

J. Sillito, K. de Voider, B. Fisher, and G. Murphy. 2005. Managing software change tasks: An exploratory study. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering, ISESE'05*. 10. DOI:http://dx.doi.org/10.1109/ISESE.2005.1541811

J. Sillito, G. C. Murphy, and K. de Volder. 2008. Asking and Answering Questions during a Programming Change Task. *Transaction in Software Engineering* 34 (July 2008), 434–451. Issue 4. DOI:http://dx.doi.org/10.1109/TSE.2008.26

J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. 1997. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 21.

L.-G. Singer. 2013. *Improving the Adoption of Software Engineering Practices Through Persuasive Interventions*. lulu.com.

D. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč. 2003. Challenges and Recommendations When Increasing the Realism of Controlled Software Engineering Experiments. In *Empirical Methods and Studies in Software Engineering*, Reidar Conradi and AlfInge Wang (Eds.). Lecture Notes in Computer Science, Vol. 2765. Springer Berlin Heidelberg, 24–38. DOI:http://dx.doi.org/10.1007/978-3-540-45143-3_3

E. Soloway and K. Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.* 10, 5 (Sept. 1984), 595–609. DOI:http://dx.doi.org/10.1109/TSE.1984.5010283

Tracy L. Tuten, David J. Urban, and Michael Bosnjak. 2002. *Internet Surverys and Data Quality: A Review*. Hogrefe and Huber Publishers.

M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 233–243. http://dl.acm.org/citation.cfm?id=2337223.2337251

A. Von Mayrhauser and A. M. Vans. 1996. Identification of Dynamic Comprehension Processes During Large Scale Maintenance. *IEEE Transactions on Software Engineering* 22, 6 (1996), 424–437. Issue 6.

J. C. F. D. Winter and D. Dodou. 2010. Five-Point Likert Items: t test versus Mann-Whitney-Wilcoxon. *Practical Assessment, Reseach and Evaluation* 15, 11 (October 2010).

PREPRINT - Authors Version - Accepted to ACM TOSEM in May 2014