What Developers (Care to) Know About Their System

Anonymous Author(s)

ABSTRACT

Developers spend most of their time with program comprehension, obtaining (or recovering forgotten) knowledge they need to perform a task. Researchers have investigated the information needs of developers to understand what knowledge is important, and to scope techniques, for example, to facilitate program comprehension, support knowledge recovery or identify experts. Similarly, researchers analyzed developers' memory, aiming to understand how developers forget—which essentially causes the need to recover knowledge. However, we are not aware of studies linking these research directions to investigate what knowledge developers keep in their mind, allowing them to ask less and different questions during knowledge recovery compared to new developers. To address this gap, we conducted an interview survey with 17 experienced developers, in which we investigated (1) what knowledge developers consider important to remember; (2) whether developers can correctly answer questions about their system from memory; and (3) how their self-assessment relates to their actual knowledge. Our results indicate, among others, that developers consider architecture and abstract code knowledge (e.g., its intent) as most important to remember, that the perceived importance relates to their ability to remember knowledge correctly, and that their self-assessment of their knowledge decreases while reflecting about their system. Based on these findings, we discuss important research directions and practical implications for managing and recovering developers' knowledge as well as for program comprehension.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Knowledge, information needs, memory, empirical study

ACM Reference Format:
Anonymous Author(s). 2020. What Developers (Care to) Know About Their System. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In their daily work, developers must constantly understand the artifacts (e.g., code) and properties (e.g., dependencies) of the system they are working on to obtain knowledge about its behavior and structure. So, a developer's knowledge impacts numerous aspects of software engineering, for example, the costs of development and maintenance, the quality of a system, and the number of bugs [2, 28, 43, 56, 60, 61]. For this reason, researchers investigate developers' understanding of a system from different perspectives, for instance, in the context of program comprehension [48, 61], information needs [9, 54] or knowledge management [6, 43]. Such works regularly propose new techniques to support developers in understanding a system, such as expertise identification [32, 33] or on-demand documentation [41, 58], and empirically assess the impact of specific development practices, such as the use of identifier names [5, 19] or code comments [7, 35].

Such research is highly valuable, helping to improve our understanding of how developers obtain knowledge about a system. Still, some fundamental questions have not been answered, potentially misguiding research to omit properly addressing needs of software developers. In this paper, we are particularly concerned with establishing a link between developers' information needs and their memory—with forgetting and software evolution arguably being the most important causes for which developers have to recover knowledge [20, 25, 37]. To the best of our knowledge, this link has not been investigated in previous research (cf. Section 2).

We investigated this link based on a two-fold empirical study. First, we reviewed the literature on questions developers ask during their tasks, adopting guidelines for systematic literature reviews [21] to improve replicability. The results of this review served as input for the design of our interview survey, and as basis for comparisons. Second, we conducted an interview survey with 17 developers. We decided for this method since the correctness of a developer's memory is difficult to assess, requiring a detailed analysis of the software system. In addition, investigating psychological aspects (i.e., memory) of human subjects poses numerous biases for quantitative studies. So, using a qualitative method promises more reliable insights [63]. In detail, we contribute:

- An analysis of empirical studies that are concerned with questions developers ask during their tasks (Section 2.2).
- A qualitative study of the knowledge developers consider important to remember (Section 4.1), the relation of importance and correctness of knowledge (Section 4.2), and developers' ability to correctly reflect on their knowledge (Section 4.3).
- Discussions of the implications for research and practice.
- An open-access repository comprising the results of our literature review, our interview guide, and anonymized results.[1]

Our results are important for research and practice alike, as we show that the information needs of developers depend on what they consider important to remember. While our interviewees could answer questions they considered less important quite well, they performed considerably better on those they perceived important. Overall, our insights imply important open research directions, and help organizations to tailor the support for program comprehension

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]Submitted as supplementary material, will be published on Zenodo

Table 1: Overview of the related work and described studies we identified during our systematic literature review.

Paper	Venue	#P	#Q	Method	S
[14]	ASE'98	–	60	Newsgroup analysis	IQ
[51]*	FSE'06	25	44	Observational study	IQ
[22]	ICSE'07	17	21	Observational study	IQ
		42	21	Survey	RI
[52]	TSE'08	*<extends Sillito et al. [51], no relevant changes>*			
[17]*	ICSE'10	11	46 (78)[a]	Interviews	IQ
[28]	ICSE'10	460	12	Survey	RD
[29]*	PLATEAU'10	179	94	Survey	IQ
[59]	FSE'12	33	8 (24)[b]	Survey	IQ
		180	15	Survey	RI
		180	15	Survey	RD
[26]	PLATEAU'14	6	7[c]	Think-aloud sessions	IQ
[36]*	VEM'14	42	11	Survey	RI
[54]*	ESEC/FSE'15	10	78 (559)[d]	Think-aloud sessions	IQ
[1]	ICST'17	194	37	Survey	IQ
[47]	CHASE'17	27	25	Survey	RI
[27]	ICPC'19	*<extends Kubelka et al. [26], no relevant changes>*			

#P: Number of Participants; #Q: Number of Questions; S: Scope
IQ: Identify Questions; RI: Rate Importance; RD: Rate Difficulty;
[a] Lists 46 questions, website not available; [b] Lists 8 questions;
[c] 7 new questions, others from previous work [22, 52]; [d] Lists 78 questions, website not available; [*] Starting points for snowballing

and knowledge recovery to the individual needs of experienced and novice developers.

2 RELATED WORK AND MOTIVATION

As time goes by, developers forget the details of the system they work on, affecting their knowledge of, for instance, source code, architecture, development history, and collaborators. Unfortunately, only few studies analyze forgetting in the context of software engineering. Kang and Hahn [20] investigated learning and forgetting curves for domain, methodological, and technological knowledge. Their results indicate that methodological knowledge is prone to forgetting, whereas domain and technological knowledge is more resilient. Complementary, Krüger et al. [25] aimed to apply and analyze a forgetting curve on code level. The results show that especially the number and ratio of contributions to the code influence developers' perception of their memory.

Due to their memory decay, software developers have to recover knowledge before working on a system again, with different types of knowledge requiring specific information, depending on the responsible part of the memory [37]. Two main directions of empirical research relate to this issue. First, researchers investigate the *information needs* of software developers during their tasks [9, 29, 36, 51, 54]. Such research aims to understand the importance of different types of knowledge for facilitating developers' tasks. Second, researchers analyze how to directly improve the *program comprehension* of a system [42, 44, 57, 61], focusing on code as the primary artifact developers inspect to understand a system [24, 35, 53, 60]. So, the goal is to aid the knowledge recovery for a system, potentially using other types of artifacts besides code as supportive means. Building on these two directions, researchers have proposed techniques, among others, to recover and visualize information [10, 12, 17, 41, 64] or to identify experts [2, 3, 18, 31, 45] that may help to understand relevant code.

2.1 Motivation

As the related work indicates, researchers have collected a large body of knowledge on information needs and knowledge recovery. In contrast, little is known about developers' memory decay, and even more important: *We are missing a link between memory decay and information needs regarding what developers consider important to remember.* This is an important link, as information may be codified in the code and additional documentation, may be easy to recover or may remain in a developer's memory for a long time. So, depending on the type and availability of knowledge (and information for its recovery), we may need specialized techniques and research to understand and facilitate practitioners' tasks.

More precisely, depending on what developers consider important to remember, different strategies to codify information and to facilitate program comprehension can be helpful. For instance, code details may not be worth remembering and are instead encoded in identifier names. Similarly, missing knowledge about (code) ownership may be encoded in a version control system, and thus is easy to recover with lightweight tooling and not worth remembering. In contrast, other knowledge may be important enough for experienced developers to remember it (e.g., memorizing a model of the system architecture). Novices are missing both, knowledge that experienced developers remember and knowledge that they recover when needed. Therefore, the information needs vary for the two groups. New developers may, for example, require additional support to gain architectural knowledge, especially if the architecture is not codified since experienced developers can remember it.

Research Questions. Based on this motivation, we aimed to investigate empirically what developers consider important to remember (**RQ$_1$**), how reliable their knowledge is (**RQ$_2$**), and how they assess their knowledge (**RQ$_3$**). This way, we intended to understand the mentioned link between memory decay and knowledge needs. Specifically, we formulated three research questions (RQs):

RQ$_1$ *What knowledge about their system do developers consider important to remember?*

RQ$_2$ *Can developers correctly answer questions about their system based on their memory?*

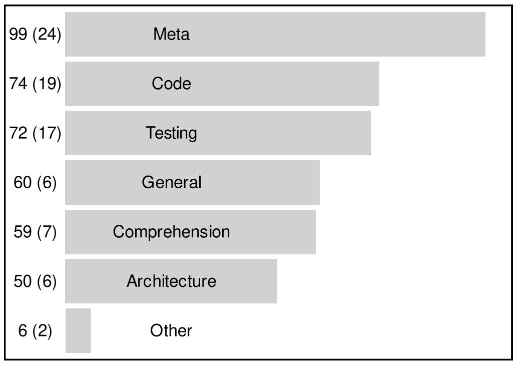
RQ$_3$ *To what extent does a developer's self-assessment align with their actual knowledge about their system?*

By answering these questions, we provide fundamentally new insights into developers' memory and information needs.

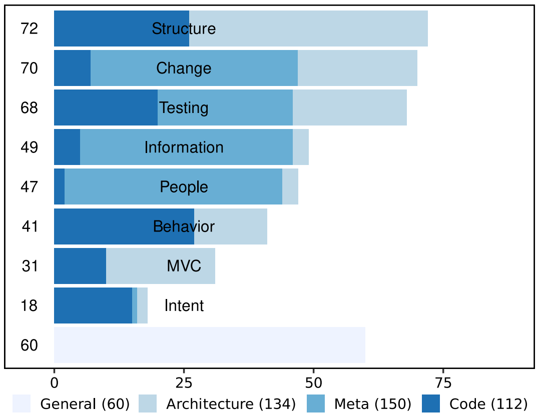
2.2 Reviewing Information Needs

We reviewed the related work describing concrete questions software developers ask during their tasks, following established guidelines for systematic literature reviews [21, 62]. This way, we aimed to i) summarize the context on what knowledge developers require, ii) ground our study design in empirical evidence, and iii) establish a dataset to compare our results with. So, our goal was not to conduct and report a full-fledged systematic literature review, which is why we adapted the following points:

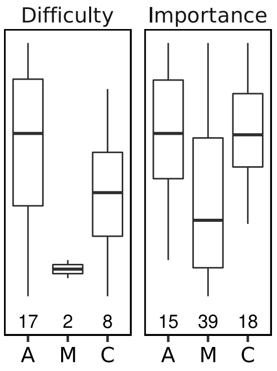
- We employed snowballing [62] based on a set of relevant papers we knew. So, we may have missed papers that we could have found with the more often used automatic search. However, automatic searches face considerable problems regarding effort and



(a) Themes identified based on the classifications defined in the selected papers. The numbers on the left indicate the numbers of questions (and classes) associated with each theme.



(b) Results of the re-classification and the number of questions associated with each theme.



(c) Relative, normalized rankings (i.e., [0,1]) of architectural (A), code (C), and meta (M) questions.

Figure 1: Overview of the themes and rankings we identified from the papers in Table 1.

replicability [21], and we argue that we used a proper starting set of papers, including papers from high quality venues.

- We did not perform a quality assessment, but trusted the review processes of the venues [8]. Also, as we aimed to classify existing findings, a quality assessment is less important [21].
- We do not report any more of the usual statistics than those in Table 1, as they are not interesting for our actual study.

These established adaptations facilitated our conduct while ensuring that we could obtain reliable data. In the following, we describe the design, conduct, and findings of our review.

Search Strategy. For our snowballing search, we started with five relevant papers (asterisked in Table 1). Then, we snowballed through references (backwards) and citing papers (forwards) to identify further papers. To be as complete as possible, we used Google Scholar for our forwards snowballing (last checked on February 11th 2020), as it indexes most papers, and did not limit the number of iterations. So, whenever we identified another relevant paper, we employed snowballing on that paper, too.

Selection Criteria. We collected empirical studies on questions developers ask, indicating missing and apparently required knowledge. Precisely, we defined the following inclusion criteria:

IC_1 The paper is written in English.
IC_2 The paper has been published at a peer reviewed venue.
IC_3 The paper reports an empirical study.
IC_4 The paper investigates development/maintenance knowledge.
IC_5 The paper does one or both of the following:
  (a) Identifies (ID) concrete questions that developers ask.
  (b) Rates the importance (RI) or difficulty (RD) of questions.

We used these criteria to ensure the quality of included papers (IC_1, IC_2) and that they provide empirical evidence (IC_3). Moreover, we excluded studies on questions too specific to ask for any subject system (IC_4), for instance, those focusing on code reviews [38], bug reports [9], API usages [13], or concurrent programming [39]. Finally, we included only papers that provide a systematic sample of concrete questions (IC_5)—excluding papers that solely exemplify some questions (e.g., Letovsky [30] or Sharif et al. [46]).

Data Extraction and Synthesis. We extracted bibliographic data, all concrete questions, and author classifications of these questions. Further, we extracted for each individual study in a paper the number of participants, the questions involved, the research method, the scope (i.e., ID, RI or RD, as defined for IC_5), and any additional comments (e.g., regarding the availability of questions). To analyze our data, we adopted card sorting [65]: First, we identified themes based on the classifications provided in the papers, aiming to unify the 81 classes (of 420 questions) defined by the authors. Second, we applied these themes to reclassify all questions (456 including those not classified by authors) based on their texts. We did this to check our first classification and gain a more detailed understanding of the questions' contexts that may be hidden by the authors' classifications. Finally, we used our re-classification to assign reported rankings of questions to themes.

Results & Discussion. We display a summary of our results in Figure 1. As we can see in Figure 1a, we initially identified seven themes based on nine classifications. During our analysis, we found that the questions of Erdem et al. [14] are *General* (e.g., "What does it do?") and can be used in any context. For this reason, we did not consider these questions in more detail in our remaining analysis. In contrast, we found that classes on *Testing*, how developers form mental models during program *Comprehension*, and *Other* issues subsume questions related to the three remaining themes. So, we decided to establish a classification based on these three themes:

(1) *Architecture* questions related to the structure of a system (e.g., "[Which] API has changed?" [17], "Who can call this?" [54]).
(2) *Meta* questions on a system's context (e.g., "Who owns a test case?" [17], "How has it changed over time?" [29]).
(3) *Code* questions on the implementation (e.g., "How big is this code?" [29], "What are the arguments to this function?" [51]).

While a suitable abstraction, we remark that questions can arguably belong to multiple themes, for example, covering the evolution (*Meta*) of a method (*Code*). For simplicity, we only assigned the theme that was predominant in a question.

To improve our understanding of questions appearing in each theme and select questions for our interviews, we investigated the actual 456 questions we extracted. In Figure 1b, we display the sub-themes we identified to specify questions in more detail. We remark that we again show only the predominant sub-theme of a question (e.g., "*Who* has made changes to [a] defect?" [17] relates primarily to *People*, but also to *Change* and *Testing*). For code and architecture, not surprisingly, the sub-themes mostly relate to a system's *Structure*, *Behavior*, the model-view-controller (*MVC*) pattern, and *Testing*. Besides testing, meta questions mostly relate to understanding a *Change*, finding *Information*, and identifying responsible *People*. Particularly on the code level, questions relate to the *Intent* of the implementation. Our re-classification is comparable to Figure 1a, yielding a similar number of questions for testing and meta information, which is the theme with most questions. Overall, we can see that fewer questions relate to the code of a system, potentially because developers can investigate it, and thus can directly recover knowledge.

Finally, in Figure 1c, we show a summary of the rankings that are reported in the identified papers. We considered our three themes and used a question's normalized, relative position (i.e., from 0 to 1) in the respective ranking, so that each question represents a value for one of the themes. As it comprises only two examples, we cannot judge the difficulty of answering meta questions, but we can still observe that the difficulty and importance of answering questions seem to relate (cf. also Tao et al. [59]). Moreover, while meta questions represent the largest theme, they seem to be considered less important than architectural or code questions. This does not seem to purely relate to the higher number of these questions. For instance, the seven (out of 21) meta questions ranked in the study of Ko et al. [22] are among the lowest nine—even though they were asked as frequently as questions from the other two themes. Overall, these insights indicate that meta questions may be frequently asked, but can be recovered more easily or are simply not important.

Insights from the Literature

• Architecture, meta, and code are general themes appearing in questions. • Difficulty and importance of the questions in a theme relate. • How often a theme's questions appear is unrelated to its importance. • Developers ask fewer questions about the source code. • Meta questions are less important. •

3 METHODOLOGY

To address our research questions, we conducted a qualitative interview survey [63], inspired by psychological and software-engineering research on forgetting [4, 11, 20, 25, 34] and our systematic literature review in Section 2.2.

3.1 Design Decisions

We intended to investigate what knowledge developers can remember and what knowledge they consider important to remember. So, a corresponding empirical study involves psychological research with human subjects, causing a variety of potential biases due to the subjects' individuality [16, 25, 50, 55]. Many of these biases are hard to impossible to control or isolate (e.g., memory performance, technological preferences, motivation). A particular issue was that we were concerned with developers' memory, and thus our subjects needed to have knowledge about the analyzed system before we conducted our study—without giving away the purpose of our study to avoid biases. As a result, we decided that we had to ask questions about one of the systems a subject worked on before. With all these issues challenging an experimental setup, we decided to employ a survey to collect qualitative and quantitative data about how developers behave [63]. More precisely, we conducted interviews to gain detailed insights, reduce faulty or empty answers, and avoid that our interviewees would drop out—considering that each interview took between 1 and 2.5 hours. To summarize, we decided to *conduct an interview survey*, asking previously identified *questions on information needs* (cf. Section 2.2) about a *subject's system*. Thus, similar to previous studies on information needs, we provide a descriptive empirical study in which we focus on obtaining a detailed understanding of a phenomenon or problem instead of simply searching for correlations.

3.2 Interview Structure

We present an overview of our semi-structured interview guide in Table 2. In the remaining paper, we use the shown identifiers to refer to the questions. Moreover, we provide one concrete source from our systematic literature review from which we adopted each question. Overall, we defined 27 questions dived into five sections, with each section having a brief introduction about its purpose and scope.

First, we were concerned with our interviewees' self-assessment of how much they still knew about their system. For a detailed overview, we asked for an assessment of the interviewee's overall system (OS_1), architectural (OS_2), meta (OS_3), and code (OS_4) knowledge. To see whether this self-assessment would change over time, for example, because an interviewee recalled more details after thinking about other questions, we repeated these questions after each of the following three sections.

For the next three sections, we adopted questions that we identified in our systematic literature review according to our themes. We selected questions that correspond to different sub-themes we identified. For instance, A_1 relates to the architectural structure of a system, M_4 to changes, and C_2 to testing. So, we aimed to design our interviews to cover a broad range of knowledge on various levels of detail, for example, C_1 to C_3 are concerned with a more abstract representation of code, while C_4 to C_6 are concerned with concrete details. Furthermore, we planned to evaluate the correctness of our interviewee's responses, wherefore we intended to include questions that we could checked against the actual system, not only our interviewees' re-evaluation.

Finally, we asked our interviewees to elaborate on our research questions. So, we asked them what knowledge they consider important to remember intuitively (IK_1), by rating our three themes (IK_2), and by rating each question individually (IK_3). Then, we asked how they reflected about their knowledge (IK_4) and for additional remarks. Afterwards, we checked the answers together with each interviewee, who are experts and could now look at their systems.

As some questions relate to concrete files or methods of the system, the interviewer prepared these questions based on the system an interviewee named at the beginning of or before the interview. To avoid biases, the interviewees were not allowed to look at any information regarding their system while we asked these questions. Note that we investigated only systems developed in a version control system, which allowed us to track precise times of an interviewee's last edit to files. So, we measured the time differences between the edit and the interview for all code

Table 2: Structure of our interview guide.

ID	S	Questions & Answers (A)
Section: Overall Self-Assessment		
<asked 4 times: at the beginning and after each of the following three sections>		
OS$_1$	–	How well do you still know your system?
OS$_2$	–	How well do you still know the architecture of your system?
OS$_3$	–	How well do you know your code of the system?
OS$_4$	–	How well do you know the file *<name>*?
		$\mathbf{A_{OS1, OS2, OS3, OS4}}$: Rating from 0 to 100 %
Section: Architecture		
A$_1$	[29]	Can you draw a simple architecture of your system?
		$\mathbf{A_{A1}}$: A drawn model *<updated after each other section>*
A$_2$	[54]	Is a database functionality implemented in your system?
A$_3$	[51]	Is a user interface implemented in your system?
		$\mathbf{A_{A2, A3}}$: ○ Yes: *<file>* ○ No
A$_4$	[28]	Can you name a file that acts as the main controller of your system?
		$\mathbf{A_{A4}}$: ○ Yes: *<file>* ○ Yes: *<functionality>* ○ No
A$_5$	[29]	On which other functionalities does the file *<file>* rely?
		$\mathbf{A_{A5}}$: Open text
A$_6$	[29]	Can you exemplify a file/functionality you implemented using a library?
		$\mathbf{A_{A6}}$: ○ Yes: *<file>* ○ Yes: *<functionality>* ○ No
Section: Meta-Knowledge		
M$_1$	[36]	When in the project life-cycle has the file *<file>* last been changed?
		$\mathbf{A_{M1}}$: Open text
M$_2$	[29]	Can you exemplify a file which has recently been changed and the reason why (e.g., last 2-3 commits)?
		$\mathbf{A_{M2}}$: ○ Yes: *<file> <reason>* ○ Yes: *<file>* ○ No
M$_3$	[17]	Can you point out an old file that has especially rarely/often been changed?
		$\mathbf{A_{M3}}$: ○ Yes: *<file>* ○ No
M$_4$	[17]	How old is this file in the project life-cycle and how often has it been changed since the creation?
M$_5$	[29]	Who is the owner of file *<file>*?
M$_6$	[29]	How big is the file *<file>*?
		$\mathbf{A_{M4, M5, M6}}$: Open text
Section: Code Comprehension		
<for three> Files: a) *<file>*; b) *<file>*; c) *<file>*		
C$_1$	[29]	What is the intent of the code in the file?
		$\mathbf{A_{C1}}$ *<per file>*: Open text
C$_2$	[29]	Is there a code smell in the code of the file?
		$\mathbf{A_{C2}}$ *<per file>*: ○ Yes: *<smell>* ○ Yes ○ No
C$_3$	[51]	Which data (in data object or database) is modified by the file?
		$\mathbf{A_{C3}}$ *<per file>*: Open text
<for three> Methods: a) *<from file a>*; b) *<from file b>*; c) *<from file c>*		
C$_4$	[29]	Which parameters does the following method need?
C$_5$	[26]	What type of data is returned by this method?
C$_6$	[51]	Which errors/exceptions can the method throw?
		$\mathbf{A_{C4, C5, C6}}$ *<per method>*: Open text
Section: Importance of Knowledge		
IK$_1$	–	Which part of your system do you consider important?
		$\mathbf{A_{IK1}}$: Open text
IK$_2$	–	Which type of the previously investigated types of knowledge do you consider important?
		$\mathbf{A_{IK2}}$: ○ Architecture ○ Meta ○ Code
IK$_3$	–	Which of the previous questions do you consider important or irrelevant when talking about familiarity?
		$\mathbf{A_{IK3}}$ *<(per A_i, M_i, and C_i)>*: ○ Irrelevant ○ Half/half ○ Important
IK$_4$	–	What do you consider/reflect about when making a self-assessment of your familiarity?
IK$_5$	–	Do you have additional remarks?
		$\mathbf{A_{IK4, IK5}}$: Open text

S: Example source identified in our systematic literature review

questions, aiming to control for the impact of time on memory. The other two themes relate to more tacit and not traceable knowledge, which is why we could not employ this analysis for these.

Table 3: Overview of the interviews in order of conduct.

ID	Area	Domain	Prog. Lang.	LOC	# D
1	Academia	Document Parser	Java	<10k	2
2	Academia	Model Editor	Java	<10k	3
3	Academia	Security Analysis	Java	<10k	1
4	Academia	Machine Learning	Python	<10k	4
5	Academia	Static Code Analysis	Java	<10k	1
6	Industry	Web Services	JavaScript, PHP	10k-100k	2
7	Industry	Web Services	PHP	>100k	1
8	Academia	IDE	Java	>100k	6
9	Academia	Databases	C++	>100k	3
10	Academia	Static Code Analysis	Java	<10k	1
11	Industry	Android App	Java	10k-100k	1
12	Industry	ERP	C#	>100k	6
13	Academia	Static Code Analysis	Java	<10k	1
14	Academia	Web Services	Ruby	<10k	1
15	Open-Source	Geometry Processing	Rust	<10k	1
16	Industry	Static Code Analysis	OCAML	<10k	2
17	Open-Source	Traceability	Java	<10k	5

# D: Number of active developers

3.3 Conduct

For conducting the interviews, at least one interviewer met with one interviewee at a time. We asked the interviewees to participate in a study on program comprehension, without exploiting our actual intent in advance. Furthermore, we asked each interviewee to provide us access (e.g., via a link in advance or by bringing their computer with the system) to their version control system. Using this access, we first prepared the actual interview guide (e.g., filling in file and method names, documenting the last time files were edited by the interviewee). Afterwards, we conducted the actual interviews, during which the interviewees could ask any question to clarify uncertainties. In the end, we checked the interviewee's answers for correctness, allowing them to also look into their system and re-evaluate what answers have been incorrect—if we could not check a question against the code or version control system.

We did not plan for a concrete number of interviews, but stopped when we found that the last three interviews did not change the average responses anymore. As described by Wohlin et al. [63], such a saturation is a reasonable stop criterion for qualitative research methods. In the end, we conducted 17 interviews. Besides reaching saturation, our sample size is comparable to those reported in similar studies on information needs (cf. Table 1), and our results yield comparable findings regarding the importance of question (compare Figure 1c and Figure 2a). Consequently, we argue that our sample size is reasonable to tackle our research questions.

3.4 Interviewees

Following the recommendation of Wohlin et al. [63], we aimed to include subjects based on differences rather than similarity, aiming to derive insights from a diverse sample. In Table 3, we provide an overview of our interviewees' systems. We remark that area refers to the system, not the interviewee—of which five worked full time in industry and four had industrial experiences from previous positions or collaborations. The systems span various domains and programming languages. Six are medium- to large-scale and the number of active developers (one system had far more than 50 contributors) ranges from one to six. Moreover, the interviewees themselves worked in different countries, for different organizations, and three are female. So, our interviewees represent a diverse sample, allowing us to generalize based on different perspectives.

3.5 Rating Correctness

We rated the correctness of answers by systematically iterating through each question together with the interviewee, and investigating the system's code as well as version control data. Assessing the correctness with the interviewee had two advantages: First, feasibility: Many of the questions can be answered only or more easily by developers who have knowledge about and access to the system. Especially for our industrial interviewees, we were not allowed to access the system alone to evaluate the answers. Second, uncertainty: We needed to asses the correctness of memory in comparison to the interviewee's individual understanding of the system. For some questions, we cannot assume that there is a single "correct" answer that would be valid from the eyes of every developer of the system. As an example, consider the intent of a file or the presence of code smells, which may be up for debate between developers of the same system.

Rating Scheme. We rated questions A_{2-6} and M_{1-6} as incorrect (0 points), partially correct (0.5 points) or correct (1 point). Identically, we rated the answers for each file and method for questions C_{1-6}. For each of these questions, the interviewee's score was then the average of the points received for the three files or methods. As example, consider the intent was correctly described for two files (1 point each) and incorrect for the third one (0 points). The interviewee's score for correctness for question C_1 was then 0.67. Note that in five cases not all questions were applicable to all files. This happened in cases where the publicly accessible repository we used for preparation was not up to date, or the chosen file did, for instance, not include a method that could be used for the interview.

A special case was rating the correctness of the architecture (A_1), as we allowed the interviewees to correct or detail the model throughout the interview. For rating, we considered three questions:

(1) Did the interviewee consider the final version of the model as correct after they had a chance to look at the system again?
(2) Was the architecture system-specific (i.e., can it be clearly associated with the system)? For instance, this was not the case for a generic drawing of a standard architecture, namely a model-view-controller without any system-specific details.
(3) Were refinements or corrections performed? Refinements were additions (e.g., of classes) made during the interview to enrich the diagram. Corrections were cases, where the interviewee removed (crossed out) or substituted parts of the model.

For architectures that the interviewee and the interviewer considered correct, system specific, and to receive only refinements (but not corrections) during the interview, we rewarded 1 point for the question. If the architecture was not system-specific, we rewarded 0 points. In one case, we rewarded 0.5 points, as the architecture was not specific, but had annotations that described the specific technologies used in the system. Finally, we rewarded 0.5 points if the architecture was system-specific, but was corrected during the interview—resulting in a correct architecture in the end.

4 RESULTS AND DISCUSSION

In this section, we present the individual results for each of our research questions. Moreover, we discuss the implications for research and practice that we derive from our observations.

4.1 RQ_1: The Importance of Knowledge

To understand what knowledge developers consider important, we analyzed the answers to questions IK_1, IK_2, and IK_3. We asked these questions after our interviewees answered all questions for which they had to remember about their system, but before assessing their correctness. More precisely, we asked about the importance of knowledge and the relation to a developer's familiarity with their system, which relates to knowledge and memory; in difference to and complementing the related work in Section 2.2.

Results. First, we extracted 35 codes from the answers to IK_1. With this analysis, we aimed to capture the knowledge our interviewees considered important based on their intuition and experience, without predefined themes. Some interviewees referred to specific classes or components of a system, but we still found common codes. In seven answers, our interviewees explicitly stated architecture as important. Closely related, we identified six codes that were concerned with dependencies, APIs, extension mechanisms, and own extensions, capturing knowledge on a system's structure. Three codes relate to the intent of the system or code as important to know, and two more referred to program comprehension or code conventions. Other codes appeared once, such as knowing bug locations, domain-specific information or the controller.

Observation 1

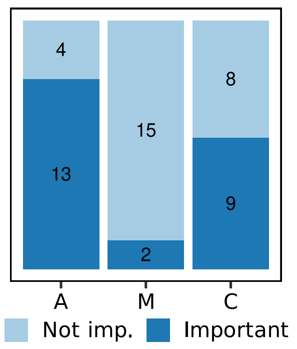
• Intuitively, most interviewees consider the architecture to be an important type of knowledge. • Some interviewees thought about dependencies and the intentions of their system. •

Second, we asked our interviewees to rate the importance of remembering knowledge based on our study. We display the corresponding results for IK_2 in Figure 2a and for IK_3 in Figure 2b. Regarding the first, high-level question (IK_2), the majority of our interviewees considered the architecture to be important. Roughly half of our interviewees stated that knowledge about the code is important, while only two argued that meta knowledge is important. Interestingly, these ratings align with the insights we identified in the literature (cf. Figure 1c), rating questions about architecture and code as more important to answer than those on meta knowledge.

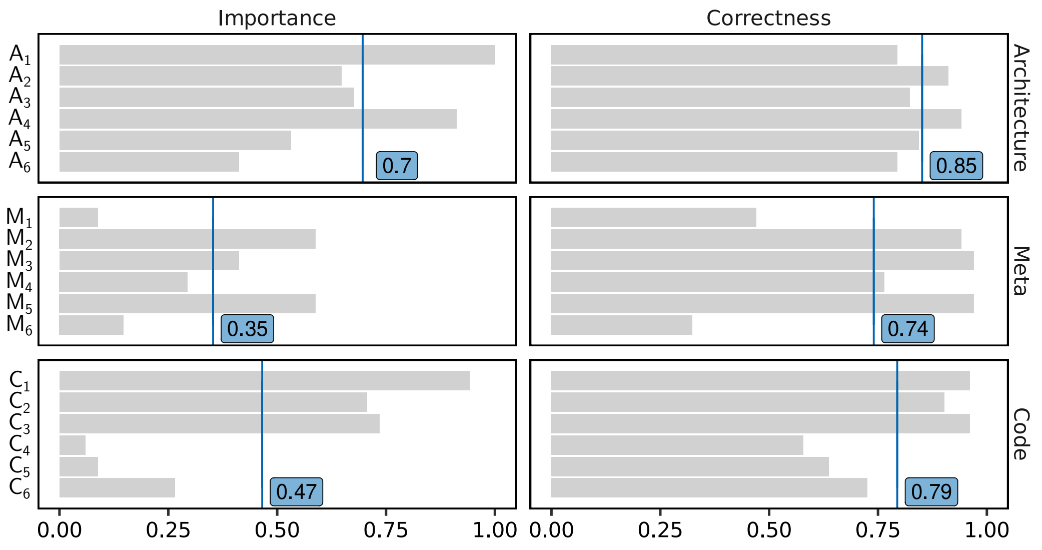
Observation 2

• Most interviewees consider architectural knowledge important to remember. • A minority of interviewees considers meta knowledge important to remember. • Half of the interviewees consider code knowledge important to remember. •

The high-level rating is reflected in the low-level ratings (IK_3), assessing the importance of remembering the knowledge to answer the questions we asked. Except A_6, all questions about the architecture were considered important by more than 50 % of our interviewees. Importantly, all interviewees agreed that it is important to be able to draw the architecture of a system (A_1). Furthermore, most interviewees agreed that it is important to know the file that serves as main controller of the system (A_4). The other concepts relating to the model-view-controller pattern, namely user interface (A_3) and data storage (A_2), were perceived similarly important. We remark that in many cases in which interviewees did not consider these two concepts important, the systems did not implement these. Slightly over half of our interviewees considered it important



(a) IK$_2$: Which type of the previously investigated types of knowledge do you consider important?



(b) Overview of our interviewees' perceived importance of each question (IK$_3$) and their correctness. The blue lines show the mean for either category and type of knowledge.

Figure 2: Our interviewees' responses regarding the importance of knowledge and their correctness when answering questions.

to know which functionality or dependencies a file relies on (A$_5$). As sole exception, less than half of our interviewees thought it is important to know files that rely on a library (A$_6$).

Observation 3

• All interviewees considered it important to know the architectural structure of their system from memory. • Most interviewees considered knowing implemented model-view-controller concepts important. •

For meta knowledge, only knowing recently changed files (M$_2$) and the owner of a file (M$_5$) were considered important by more than half of our interviewees. Some participants considered it important to recall old files that are rarely or often changed (M$_3$, M$_4$). The ability to remember how big a file is (M$_6$) or when it was last changed (M$_1$) were considered to be not important by most interviewees. Interestingly, while the averages for architecture and code knowledge align well with the overall assessment and related work, the average importance of meta knowledge is considerably higher for the questions. When asked in general, only two interviewees considered it important to know meta information. However, questions M$_{2-6}$ were all rated important by more than two interviewees.

Observation 4

• Meta knowledge regarding recent changes and ownership were considered comparably important. •

In Figure 2b, we can see that code questions can be split into two groups regarding their importance for knowledge. More than 70 % of our interviewees considered it important to be able to tell from memory what the intent of a file is (C$_1$), whether and what data is manipulated in a file (C$_3$) or whether a file contains code smells (C$_2$). In contrast, few interviewees considered it important to know details about single methods (C$_{4-6}$). However, even on this level of detail, the question asking for exceptions (C$_6$), and thus relating to quality and testing, received a comparably high score.

Observation 5

• Most interviewees considered it important to recall code knowledge concerning intent, data manipulation, and quality. • Few interviewees considered it important to know method and code details. •

Discussion. Aligning our observations with the related work, we can derive the following implications:

⇒ Architectural knowledge is important to remember and corresponding questions are difficult to answer (cf. Figure 1c). This indicates that architectural knowledge may often be tacit, causing information needs when details are forgotten or new developers work on a system. So, *research* on recovering architectural knowledge is an important direction, while documenting a system's architecture is essential for *practice*.

⇒ While meta knowledge is often considered to be less important in general, we found a discrepancy when comparing this to the importance of concrete questions. Arguably, this finding may be related to the small number of systems we investigated that are developed in collaboration, but we also identified this insight from the related work (cf. Figure 1c). For *research*, this observation implies the need to better understand the importance of meta knowledge in different contexts.

⇒ Code knowledge is considered important, but less than architecture knowledge. A detailed analysis revealed that this situation may relate to the different abstraction levels of knowledge. Developers seem to consider it important to know the intent or design flaws of the code, while they are not interested in remembering code details (e.g., parameters) that are directly encoded in the code. For *research*, this indicates the importance of two directions: developing techniques to recover the intent of source code and empirically investigating program comprehension. In *practice*, our observations implicate that improving code quality as well as documenting and tracing the intent, features, and bugs of code is essential.

So, to answer **RQ$_1$**, we find that more abstract knowledge is considered more important to remember. Particularly, developers seem to memorize knowledge about a system's architecture and the intent of its code. In contrast, more detailed knowledge—which is relevant for specific tasks, is encoded in the code, and may be well supported by improving program comprehension; as well as meta knowledge—which may be easily recoverable from version control systems; are considered less important to remember.

4.2 RQ$_2$: Correctness and Importance

Next, we compare how important our interviewees' considered it to know the answer to a question to the correctness of their answers.

Results. In Figure 2b we show the average correctness of our interviewees' answers per question. They did generally well on all questions, with an overall average correctness of 80 %. Each architectural question was answered correctly by at least 79 % of our interviewees. We can see the highest correctness for naming a file that acts as controller (A$_4$) and for pointing to the data storage (A$_2$).

Regarding meta knowledge, three questions reach an average correctness of more than 80 %. These questions are: Naming an old file that has been changed particularly rarely or often (M$_3$), naming a file that has recently been changed (M$_2$), and naming the owner of a file (M$_5$). We have to be careful with interpreting these results, as many of the systems we investigated have only one developer—in larger teams the correctness would potentially be lower. Interestingly, the only two questions that less than half of our interviewees could answer correctly are also related to meta knowledge. Particularly, the questions for the size of a file (M$_6$) and the last time a file has been changed (M$_1$).

Finally, the questions about code resemble the previous pattern of comprising two different groups. Questions about method details have been answered with an average correctness of 55 % to 75 %. Of these, the question easiest to answer seems to be the one regarding errors and exceptions (C$_6$). Again, this may be caused by our sample, as most methods we studied did not throw any exceptions. In contrast, questions about the intent of code files (C$_1$), the data manipulated (C$_3$), and the presence of code smells (C$_2$) are among the questions for which our interviewees performed best.

To investigate whether this observation is meaningful, we show the differences in correctness of our interviewees regarding the two groups and their combination in Figure 3—also considering the time between the interviewee's last edit and interview in days. Note that we removed five entries (cf. Section 3.5): We could not completely rate two cases and in three cases the commit histories were lost due to a repository migration. This resulted in 46 paired data points. We can see that, independently of the time, the more abstract questions (i.e., "File"), were answered correctly more often.

To test, using the R environment [40], whether this observable difference may be significant, we first tested whether the results are independent of the time. Using Kendall's τ, which is a comparably strict rank correlation (compared to Spearman's ρ) that we used because it allows to test non-normal distributions (i.e., correctness), we find no relevant ($-0.136<\tau<0.005$) or significant (p-value>0.05) correlation between time and correctness for any group (the method group has the negative tendency of -0.136). This indicates that the difference we can see may depend solely on the type of knowledge.

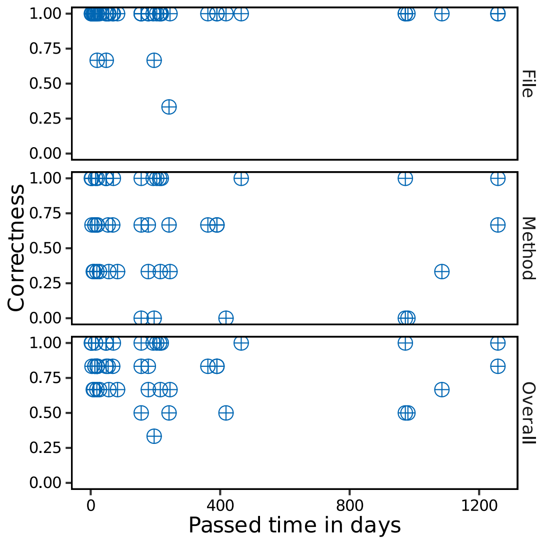


Figure 3: Comparison of interviewees' correctness on code questions (File: C$_{1–3}$; Method: C$_{4–6}$; Overall: C$_{1–6}$).

We tested this hypothesis using the Wilcoxon signed-rank test for not normally distributed and paired data to compare the means of both correctness distributions. The results indicate that the different groups of code knowledge lead to significantly different results for correctness (p-value<0.001).

Observation 6

• The interviewees achieve a high correctness (80 %) when answering questions about their systems from memory. • On average, architecture questions are answered correctly most often. • The questions most often answered correctly are concerned with the intent of code, data modified in code, owners of files, files that rarely/often changed, recent changes, and the main controller of the system. • Questions on code abstractions (e.g., intent) are answered correctly significantly more often than those on code details. •

In Figure 2b, we can see that the correctness of the answers resembles the importance of questions. However, the averages of correct answers are far closer to each other than those of importance. Also, we can see that only comparably unimportant questions resulted in fewer correct answers. More precisely, none of the questions reported important by more than half of our interviewees received less than 75 % correct answers. To test whether our observation that importance and correctness may correlate (i.e., important questions may be harder to check or are kept in mind), we again employed Kendall's τ. It reveals a significant, moderately positive correlation between both aspects (τ=0.508, p-value<0.005) for a confidence interval of 0.95. Due to the qualitative nature of our study, we obtained only few data point—wherefore the statistical power of this test is low (0.575). Still, it is a supportive indicator for our observation that developers seem to perform well at remembering knowledge they consider important (or forgetting the unimportant knowledge).

Observation 7

• Our interviewees perform better at remembering knowledge they consider important. •

Discussion.. We can derive the following implications:

⇒ The importance of knowledge and developers' ability to remember it seem to relate. Considering *research*, this highlights an important direction to investigate, as different developers (e.g., experts, novices) require tailored support during program comprehension. For instance, our results suggest that particularly code-level program comprehension is important to support, as developers do not remember code details. In contrast, architectural knowledge and the intent of code are important, especially for onboarding novices, and system experts have the tacit knowledge. In *practice*, our results indicate that developers who are experienced with a system are arguably the best support to onboard new developers and provide a general overview of the system and its intent. However, on code level and for meta knowledge (e.g., responsibilities), technical support and documentation could help all developers.

⇒ In contrast to Krüger et al. [25], we found no correlation (but a non-significant tendency) between the time that has passed and developers' knowledge about code. However, this analysis was not our main goal (we focused on knowledge types) and there are essential design differences between both studies. Nonetheless, our results are interesting and ask for additional *research* on developers' ability to remember different types of knowledge over time (similar to Kang and Hahn [20]).

⇒ An important *research* problem that we indicated in Section 2.2 and found again, is the question what the importance of a question (or knowledge) actually implies? Apparently, developers do remember the knowledge to answer important questions better. However, they may simply be important because it is challenging to check them or not easy to recover with existing tools. Tao et al. [59] investigated this problem and identified some of the issues, but they focus solely on code changes and do not investigate developers' memory.

So, to answer **RQ$_2$**, we find that developers perform quite well in answering questions about their systems, even for questions they do not consider important. However, our results indicate that the perceived importance of knowledge has a positive impact on whether developers remember it or not.

4.3 RQ$_3$: Self-Assessment and Correctness

Finally, we investigated how well the self-assessments (OS$_{1-4}$) of our interviewees resemble their ability to remember. Moreover, we investigate how they reflected about their knowledge to derive their self-assessment (IK$_4$).

Results. In Figure 4, we compare the interviewees' overall self-assessment (OS$_1$) and the correctness of their answers. We show only the overall as well as initial and final assessment because they summarize the other self-assessments. Interestingly, only one interviewee increased their assessment (+5 %), while eight kept it as it was. Most interviewees actually decreased their assessment (-13.75 % on average). Note that we cannot discuss whether a self-assessment is a perfect prediction of the correctness of answers

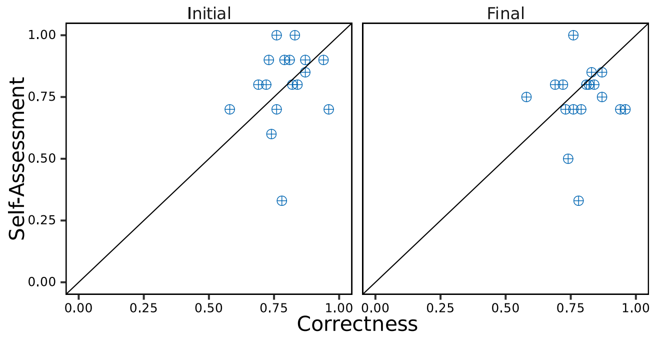


Figure 4: Averaged correctness of each interviewee compared to their overall self-assessment (OS$_1$).

given to our questions, as changing these questions could easily lead to different results. Nonetheless, it is interesting to see that the initial self-assessment included roughly a similar number of cases with a nominal value below and above the participants correctness. However, for the final self-assessment, we can see a drop that leads to most nominal values of the self-assessment being below the actual correctness. Interestingly, this suggests that the initial self-assessment seems more reliable considering a developers' knowledge than an assessment after reflecting about their system.

Observation 8

• Our interviewees' self-assessment did not change heavily during the interviews. • The initial self-assessment seems to be closer to the participants' performance compared to the final self-assessment. •

Existing research assumes a correlation between self-assessments and a developer's knowledge or experiences (depending on certain additional factors, for instance, educational degree and expertise), using self-assessments in guidelines and studies [15, 23, 25, 49]. To test this assumption, we used Kendall's τ on our data. The test results show no significant correlations (p-values>0.05) and only a small positive tendency (initial τ=0.176, final τ=0.032). However, this outcome is not surprising, as we can see in Figure 4 that our data is based on only 17 data points. It is important to note that the correlation test with 17 participants was only designed to show a strong correlation with a sufficient power (80%), but not medium or small ones. Thus, we cannot find evidence for the assumptions in previous works, indicating the need for more detailed analyses.

Observation 7

• There is no significant correlation (only a tendency) between our interviewees' self-assessments and correctness. •

In the end, we analyzed our interviewees' qualitative responses (IK$_4$) to understand how they derived their self-assessments. Our interviewees stated a variety of aspects they considered, but we could identify reappearing themes. Most often mentioned were reflections about the general structure and architecture of the system (eight times). Five interviewees reflected about the work they did on the system. Other aspects we found were the memory of efforts put into the system, the intent of the system or consideration of details. Some interviewees compared their memory to the level of understanding that they have for another system they are currently working on or to the best understanding they had about their system at any point in time. Two interviewees reported that they thought about how long it would take them to start modifying the system again. Four interviewees relied on their gut feeling and just guessed their level of knowledge—which undermines the aforementioned assumption of self-assessments being a reasonable measure to some extent. Finally, some interviewees stated that their changed self-assessment during the interview was caused by the feeling that the questions revealed gaps in their knowledge. Note that the main overlap between these results and the perceived importance of knowledge is in the architecture. Other aspects, such as previous work done, could be considered meta-knowledge, which is interestingly an area that is perceived as less important by participants.

Observation 8

• Our interviewees reflected about various aspects, mainly architecture, effort, and intent, to assess their knowledge. • Some of our interviewees simply guessed their self-assessment. •

Discussion. Building on our observations, we can derive:

⇒ Interestingly, the initial self-assessment of our interviewees' was on average closer to their actual knowledge than the final one. Also, while only few interviewees adopted their assessments, most reduced their score, resulting in a more negative perception of their knowledge. This is an interesting observation causing important implications for research, even though we found only a tendency, but no correlation due to the qualitative nature of our study. Some of our interviewees stated to simply have guessed their self-assessment, challenging the reliability of using self-assessments without control. So, *research* has to investigate in more detail to what extent what self-assessments are reasonable to rely on, for example, considering guidelines and recommendations for research methods. In addition, it is interesting to understand why developers' may underestimate their knowledge after (correctly) reflecting about it.

⇒ We found interesting aspects that our interviewees reflected about when assessing their knowledge about their systems. For example, we are not aware of tools for expertise identification that consider the actual efforts spent on a system or the knowledge of a system's intent. So, considering *research* and *practice*, our results imply additional opportunities to assess the expertise of developers and support their knowledge recovery. A particular open question is how efforts spent on a program relate to the effort of working on it again, and investigating the impact of different types of knowledge on these efforts.

So, to answer **RQ$_3$**, we find that the initial self-assessment of developers seems to align better to their knowledge compared to assessments after they reflected about their knowledge. However, we could not identify a correlation between self-assessments and correctness. Finally, we identified aspects that developers reflect about when performing self-assessments, indicating opportunities for extending existing techniques and tools.

5 THREATS TO VALIDITY

Internal Validity. Our study was concerned with psychological aspects and other human factors, namely developers' memory, their knowledge, and opinions. So, there are numerous background factors, such as age, gender, motivation or a subject's memory performance, that we cannot control—and that certainly intervene with the factors we wanted to observe. We aimed to mitigate this threat by conducting a qualitative study with a diverse sample of subjects to collect qualitative data and detailed insights from different perspectives [63]. Another threat to the internal validity are the questions we used. They may have not been ideal to achieve the goal of our study or could be misunderstood. We limited this threat by grounding the question selection in empirical evidence and conducting face-to-face interviews, allowing to clarify confusions.

External Validity Due to the qualitative nature of our study and its small sample size (compared to quantitative studies), the external validity of our results is threatened. We aimed to mitigate this threat by involving diverse developers (e.g., different countries, domains, systems), looking for saturation, and comparing our findings against related work. While these measures indicate that the external validity of our study is not compromised, the threats remain. So, additional and replication studies, also involving quantitative methods, can help to overcome this threat. Still, as we conducted a qualitative study and carefully analyzed our data, we argue that our insights provide important and reliable insights.

Conclusion Validity. We reviewed the literature to identify questions for our interview survey and re-classified the questions based on themes we identified. So, other researchers would maybe derive different classifications, and thus the results would change. This threatens the conclusions we derived for these classifications, but we analyzed the individual questions, obtained similar results as the papers we reviewed, and cross-checked our results—mitigating this threat. Moreover, we followed guidelines [21, 62, 63, 65] for our research methods to ensure that we derived meaningful results. Finally, we make all data we collected in this study publicly available to enable replications and allow others to evaluate our findings.

6 CONCLUSION

In this paper, we investigated what knowledge developers consider important to remember, whether they can correctly answer questions about their system from memory, and whether their self-assessment aligns with the correctness. For this purpose, we reviewed the literature to capture the state of the art on information needs and to design an interview survey. Based on interviews with 17 experienced developers, we found:

- Developers consider abstract knowledge more important to remember, such as a system's architecture and code intentions.
- Developers achieve a higher correctness from their memory regarding the questions they consider important.
- Developers' self-assessments may be reliable, but their assessments decreases while reflecting about their system.

To the best of our knowledge, we are the first to investigate the link between developers' memory and their information needs, considerably extending the existing body of knowledge and providing important insights for research and practice. We highlighted several research opportunities, for instance, adopting knowledge-recovery techniques according to our findings and investigating the reliability of self-assessments. In future work, we intend to extend and replicate this study, involving more subjects, additional research methods, and more detailed analyses of types of knowledge.

REFERENCES

[1] Abdullah Al-Nayeem, Krzysztof Ostrowski, Sebastian Pueblas, Christophe Restif, and Sai Zhang. 2017. Information Needs for Validating Evolving Software Systems: An Exploratory Study at Google. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 544–545.

[2] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering*. 361–370.

[3] Guilherme Avelino, Leonardo Passos, Fabio Petrillo, and Marco Tulio Valente. 2018. Who Can Maintain This Code?: Assessing the Effectiveness of Repository-Mining Techniques for Identifying Software Maintainers. *IEEE Software* 36, 6 (2018), 34–42.

[4] Lee Averell and Andrew Heathcote. 2011. The form of the forgetting curve and the fate of memories. *Journal of Mathematical Psychology* 55, 1 (2011), 25–35.

[5] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.

[6] Finn Olav Bjørnson and Torgeir Dingsøyr. 2008. Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. *Information and Software Technology* 50, 11 (2008), 1055–1068.

[7] Jürgen Börstler and Barbara Paech. 2016. The role of method chains and comments in software readability and comprehension—an experiment. *IEEE Transactions on Software Engineering* 42, 9 (2016), 886–898.

[8] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software* 80, 4 (2007), 571–583.

[9] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2009. *Frequently Asked Questions in Bug Reports*. Technical Report. University of Calgary.

[10] Christopher S Campbell, Paul P Maglio, Alex Cozzi, and Byron Dom. 2003. Expertise identification using email communications. In *Proceedings of the twelfth international conference on Information and knowledge management*. 528–531.

[11] Gillian Cohen and Martin A Conway. 2007. *Memory in the real world*. Psychology Press.

[12] Brian De Alwis and Gail C Murphy. 2008. Answering conceptual queries with ferret. In *Proceedings of the 30th international conference on Software engineering*. 21–30.

[13] Ekwa Duala-Ekoko and Martin P Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 266–276.

[14] Ali Erdem, Lewis Johnson, and Stacy Marsella. 1998. Task Oriented Software Understanding. In *International Conference on Automated Software Engineering*. IEEE, 230–239.

[15] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 73–82.

[16] Robert Feldt, Lefteris Angelis, Richard Torkar, and Maria Samuelsson. 2010. Links between the personalities, views and attitudes of software engineers. *Information and Software Technology* 52, 6 (2010), 611–624.

[17] Thomas Fritz and Gail C. Murphy. 2010. Using Information Fragments to Answer the Questions Developers Ask. In *International Conference on Software Engineering (ICSE)*. ACM, 175–184.

[18] Thomas Fritz, Gail C Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. 2014. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 2 (2014), 1–42.

[19] Johannes C Hofmeister, Janet Siegmund, and Daniel V Holt. 2019. Shorter identifier names take longer to comprehend. *Empirical Software Engineering* 24, 1 (2019), 417–443.

[20] Keumseok Kang and Jungpil Hahn. 2009. Learning and Forgetting Curves in Software Development: Does Type of Knowledge Matter?. In *International Conference on Information Systems (ICIS)*. Association for Information Systems, 194:1–194:15.

[21] Barbara Ann Kitchenham, David Budgen, and Pearl Brereton. 2015. *Evidence-based software engineering and systematic reviews*. Vol. 4. CRC press.

[22] Andrew J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 344–353.

[23] Andrew J Ko, Thomas D Latoza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.

[24] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet 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 (2006), 971–987.

[25] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do you remember this source code?. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 764–775.

[26] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. 2014. Asking and Answering Questions during a Programming Change Task in Pharo Language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools* (Portland, Oregon, USA) *(PLATEAU '14)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2688204.2688212

[27] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2019. Live programming and software evolution: questions during a programming change task. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 30–41.

[28] Thomas D. LaToza and Brad A. Myers. 2010. Developers Ask Reachability Questions. In *International Conference on Software Engineering (ICSE)*. ACM, 185–194.

[29] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-Answer Questions about Code. In *Evaluation and Usability of Programming Languages and Tools* (Reno, Nevada) *(PLATEAU '10)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/1937117.1937125

[30] Stanley Letovsky. 1987. Cognitive Processes in Program Comprehension. *Journal of Systems and software* 7, 4 (1987), 325–339.

[31] Shuyi Lin, Wenxing Hong, Dingding Wang, and Tao Li. 2017. A survey on expert finding techniques. *Journal of Intelligent Information Systems* 49, 2 (2017), 255–279.

[32] David W McDonald and Mark S Ackerman. 2000. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 231–240.

[33] Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 503–512.

[34] Tim P Moran. 2016. Anxiety and working memory capacity: A meta-analysis and narrative review. *Psychological Bulletin* 142, 8 (2016), 831.

[35] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting source code: is it worth it for small programming tasks? *Empirical Software Engineering* 24, 3 (2019), 1418–1457.

[36] Renato Novais, Creidiane Brito, and Manoel Mendonça. 2014. What Questions Developers Ask During Software Evolution? An Academic Perspective. In *Workshop on Software Visualization, Evolution, and Maintenance (VEM)*. 14–21.

[37] Chris Parnin and Spencer Rugaber. 2012. Programmer information needs after memory failure. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 123–132.

[38] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–27.

[39] Gustavo Pinto, Weslley Torres, and Fernando Castor. 2015. A study on the most popular questions about concurrent programming. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. 39–46.

[40] R Core Team. [n.d.]. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org

[41] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. 2017. On-demand developer documentation. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 479–483.

[42] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 255–265.

[43] Ioana Rus, Mikael Lindvall, and S Sinha. 2002. Knowledge management in software engineering. *IEEE software* 19, 3 (2002), 26–38.

[44] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 308–311.

[45] David Schuler and Thomas Zimmermann. 2008. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*. 121–124.

[46] Khaironi Y Sharif, Michael English, Nour Ali, Chris Exton, JJ Collins, and Jim Buckley. 2015. An empirically-based characterization and quantification of information seeking through mailing lists during open source developers' software evolution. *Information and Software Technology* 57 (2015), 77–94.

[47] Vibhu Saujanya Sharma, Rohit Mehra, and Vikrant Kaulgud. 2017. What do developers want? an advisor approach for developer priorities. In *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 78–81.

[48] Janet Siegmund. 2016. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. IEEE, 13–20.

[49] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19, 5 (2014), 1299–1334.

[50] Janet Siegmund and Jana Schumann. 2015. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering* 20, 4 (2015), 1159–1192.

[51] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask During Software Evolution Tasks. In *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 23–34.

[52] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2008. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering* 34, 4 (2008), 434–451.

[53] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*. 174–188.

[54] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 248–259.

[55] Webb Stacy and Jean MacMillan. 1995. Cognitive bias in software engineering. *Commun. ACM* 38, 6 (1995), 57–63.

[56] Thomas A Standish. 1984. An essay on software reuse. *IEEE Transactions on Software Engineering* 5 (1984), 494–497.

[57] M-A Storey. 2005. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE, 181–191.

[58] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. 643–652.

[59] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[60] Rebecca Tiarks. 2011. What maintenance programmers really do: An observational study. In *Workshop on Software Reengineering*. Citeseer, 36–37.

[61] Anneliese Von Mayrhauser and A Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.

[62] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 1–10.

[63] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[64] Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C Gall. 2010. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 165–174.

[65] Thomas Zimmermann. 2016. Card-sorting: From text to themes. In *Perspectives on Data Science for Software Engineering*. Elsevier, 137–141.