What Developers (Care to) Recall: An Interview Survey on Smaller Systems

Jacob Krüger
University of Toronto, Canada
Otto-von-Guericke University Magdeburg, Germany
Email: jkrueger@ovgu.de

Regina Hebig
Chalmers | University of Gothenburg, Sweden
Email: hebig@chalmers.se

***Abstract*—Developers spend most of their time with program comprehension, obtaining (or recovering) the 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 to understand how they 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 aim to keep in their memory, allowing them to ask less and different questions during knowledge recovery. 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 recall knowledge about their (smaller) systems; 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 recall knowledge correctly, and that their self-assessment decreases while reflecting about their system. Based on these findings, we discuss research directions and practical implications for managing and recovering developers' knowledge.**

***Index Terms*—Knowledge, information needs, memory**

I. 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, development and maintenance costs, system quality, and the number of bugs [2], [25], [30], [45], [59], [63], [64]. For this reason, researchers investigate developers' understanding of a system from different perspectives, for instance, in the context of program comprehension [46], [51], [64], information needs [9], [57], or knowledge management [6], [45]—regularly proposing new techniques to support developers, for example, with expertise identification [34], [35] or on-demand documentation [43], [61]. Moreover, empirical studies assess the impact of specific practices, such as the use of identifier names [5], [19] or code comments [7], [37], on program comprehension.

Such research is highly valuable since it improves our understanding of how developers obtain knowledge about a system. In particular, forgetting and software evolution are among the most important causes for which developers' knowledge becomes less reliable [20], [27], [39], and for which they need to recover knowledge. Still, fundamental questions about the link between developers' memory and their information needs remain open. For instance, it is unclear what information developers consider important to, and actually do, remember about their systems. That, however, impacts their information needs and how respective tools should be designed.

We investigated this link based on a two-fold study. First, we reviewed the literature on questions developers ask during their tasks, adopting guidelines for systematic literature reviews (SLRs) [21]. The results of our SLR served as input for the design of an interview survey, and as basis for comparisons. Second, we conducted an interview survey with 17 developers who mostly develop smaller systems. We decided for this method since the correctness of a developer's memory is difficult to assess and subject to several threats, due to its subjective and psychological nature [27], [58]. So, using a qualitative method promises more reliable insights [66]. In detail, we contribute:

- An analysis of empirical studies that are concerned with questions developers ask during their tasks (Sec. II-B).
- Insights on the knowledge developers consider important to remember (Sec. IV-A), the relation of importance and correctness of knowledge (Sec. IV-B), and developers' ability to correctly recall their knowledge (Sec. IV-C).
- Discussions of the implications for research and practice.
- An open-access repository with the results of our SLR, our interview guide, and anonymized results.[1]

Our results show that the information needs of developers depend on what they consider important to remember; and, not surprisingly, that meta knowledge is less important for smaller systems with few developers. Overall, our insights imply important research directions, and can help to tailor the support for program comprehension and knowledge recovery.

II. Related Work and Motivation

As time goes by, developers forget the details of systems they work on, affecting their knowledge of, for instance, source code, architecture, evolution, and collaborators. Unfortunately, only few studies analyze forgetting in software engineering. Kang and Hahn [20] investigated learning and forgetting curves for domain, methodological, and technological knowledge. Their

[1] https://doi.org/10.5281/zenodo.3972404

TABLE I: Overview of the related work.

Paper	Venue		#P	#Q	Method	S
[14]	ASE'98		—	60	Newsgroup analysis	IQ
[54]*	FSE'06		25	44	Observational study	IQ
[22]	ICSE'07	{	17	21	Observational study	IQ
			42	21	Survey	RI
[55]	TSE'08		<*extends Sillito et al. [54], no relevant changes*>			
[17]*	ICSE'10		11	46 (78)[a]	Interviews	IQ
[30]	ICSE'10		460	12	Survey	RD
[31]*	PLATEAU'10		179	94	Survey	IQ
[62]	FSE'12	{	33	8 (24)[b]	Survey	IQ
			180	15	Survey	RI
			180	15	Survey	RD
[28]	PLATEAU'14		6	7[c]	Think-aloud sessions	IQ
[38]*	VEM'14		42	11	Survey	RI
[57]*	ESEC/FSE'15		10	78 (559)[d]	Think-aloud sessions	IQ
[1]	ICST'17		194	37	Survey	IQ
[50]	CHASE'17		27	25	Survey	RI
[29]	ICPC'19		<*extends Kubelka et al. [28], 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], [55]; [d] Lists 78 questions, website not available; * Starting points for snowballing

results indicate that methodological knowledge is prone to forgetting, whereas domain and technological knowledge is more resilient. Complementary, Krüger et al. [27] 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 [39]. Two main directions of empirical research relate to this issue. First, researchers investigate the *information needs* of software developers during their tasks [9], [31], [38], [54], [57]. 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 [44], [46], [60], [64], focusing on code as the primary artifact developers inspect to understand a system [24], [37], [56], [63]. The goal is to aid knowledge recovery, 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], [43], [67] or to identify experts [2], [3], [18], [33], [47].

A. Motivation

As the related work shows, researchers have collected a large body of knowledge on information needs and 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 developers' tasks.

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. Also, novices are missing both, knowledge that experienced developers remember and knowledge they can recover when needed. Therefore, the information needs vary between both 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. Advancing on existing research, we investigate particularly such links between developers' memory and their information needs.

Research Questions (RQs). Building on this motivation, we aimed to understand what developers consider important to remember (**RQ$_1$**), how reliable their knowledge is (**RQ$_2$**), and how they assess their knowledge (**RQ$_3$**). Specifically, we formulated three research questions:

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.

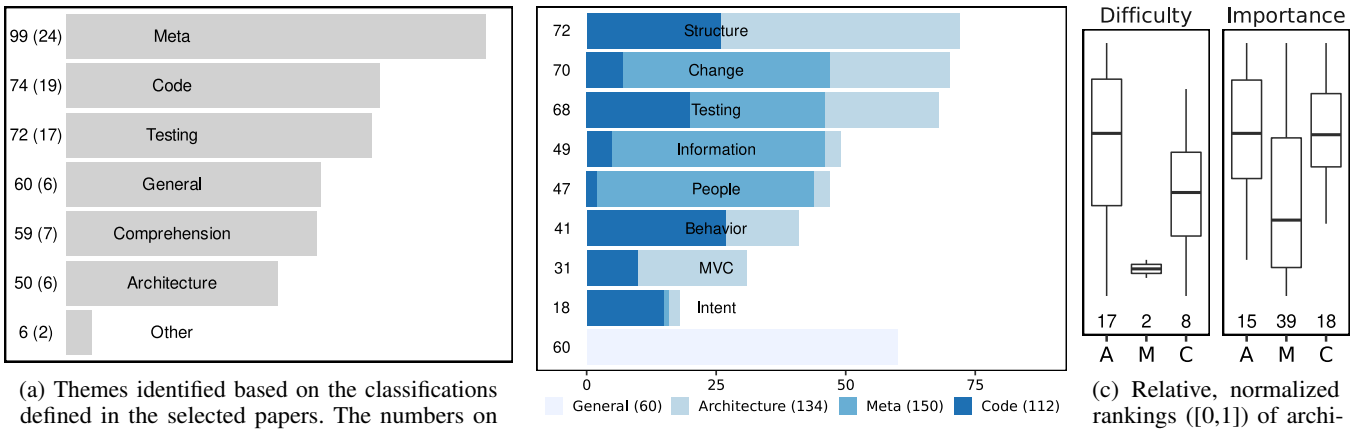
B. Reviewing Information Needs

We reviewed the related work on questions software developers ask during their tasks, following established guidelines for SLRs [21], [65]. This way, we aimed to i) summarize the research on information needs, 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 a full-fledged SLR, which is why we adapted the following points:

- We employed snowballing [65] based on a set of relevant papers we knew. So, we may have missed papers that we could have found with an automatic search. However, automatic searches face considerable problems regarding effort and replicability [21], [26], [48], and we argue that we used a proper starting set of papers from appropriate venues.
- We did not perform a quality assessment, but trusted the review processes of the venues [8]. Also, since we classify existing findings, a quality assessment is less important [21].
- We do not report the typical statistics of an SLR (besides those in Tbl. I), because they are not relevant for our study.

These established adaptations facilitated our conduct, while ensuring that we could obtain reliable data.

Search Strategy. For our snowballing search, we started with five relevant papers (asterisked in Tbl. I). 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 11$^{\text{th}}$ 2020), as it indexes most papers, and did not limit the number of iterations. So, whenever we identified a relevant paper, we employed snowballing on that paper, too.



(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 ([0,1]) of architectural (A), meta (M), and code (C) questions.

Fig. 1: Overview of the themes and rankings we identified from the papers in Tbl. I.

Selection Criteria. We collected 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 analyzes development/maintenance questions.
IC_5 The paper does one or both of the following:
  a) Identifies (ID) concrete questions that developers ask.
  b) Rates 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 [40], bug reports [9], API usages [13], or concurrent programming [41]. Finally, we included only papers that provide a systematic sample of concrete questions (IC_5)—excluding papers that only exemplify some questions (e.g., by Letovsky [32] or Sharif et al. [49]).

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 – cf. IC_5), and any additional comments (e.g., regarding the availability of questions). To analyze our data, we adopted card sorting [68]: 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 (i.e., architecture, meta, code) to synthesize the rankings of questions in the papers. For example, if the ranking in a paper was not normalized, we took the ranking of a question (e.g., #2 out of 21) and normalized that ranking (i.e., 0.95). Then, we averaged the rankings of all questions in a theme to synthesize the rankings of all papers.

Results & Discussion. We display a summary of our results in Fig. 1. As we can see in Fig. 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:

A *Architecture* questions are concerned with the structure of a system (e.g., "[Which] API has changed?" [17], "Who can call this?" [57]).
M *Meta* questions on a system's context (e.g., "Who owns a test case?" [17], "How has it changed over time?" [31]).
C *Code* questions about the implementation of a system (e.g., "How big is this code?" [31], "What are the arguments to this function?" [54]).

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

To improve our understanding of the questions and select questions for our interviews, we investigated all 456 questions. In Fig. 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 (which we considered separately, due to its apparent importance), 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 Fig. 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 the corresponding knowledge.

Finally, in Fig. 1c, we summarize the rankings 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 of each paper. Since we found 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 (as also found by Tao et al. [62]). 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 appears seems unrelated to its importance. • Developers ask fewer questions about source code. • Meta questions seem less important. •

III. Methodology

To address our research questions, we conducted a qualitative interview survey [66], inspired by research on forgetting [4], [11], [20], [27], [36] and our SLR (cf. Sec. II-B).

A. Design Decisions

We intended to investigate what knowledge developers can recall 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], [27], [53], [58]. Many of these biases are hard to impossible to control or isolate (e.g., memory performance, 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 study's purpose 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 [66]. More precisely, we conducted interviews to gain detailed insights, reduce faulty or empty answers, and limit the drop out rate (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* 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 searching for correlations.

B. Interview Structure

We display an overview of our semi-structured interview guide in Tbl. II. In the remaining paper, we use the identifiers to refer to the questions. Moreover, we provide one concrete source from our SLR from which we adopted each question. Overall, we defined 27 questions divided 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.

In the next three sections, we adopted questions from our SLR according to our (sub-)themes. 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 evaluated the correctness of our interviewees' responses for these questions. Therefore, we included questions that could be quantified against the actual system.

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 evaluated the answers together with each interviewee, who are experts and could now look at their systems.

Since our study focuses solely on memory, we decided to *not* allow developers to look at their code during the interview, as that would initiate program comprehension. Furthermore, we are aware that the order and wording of questions can be problematic for this type of study. To mitigate potential confirmation biases, we decided to ask about importance at the end (but before checking correctness) to avoid that our interviewees may focus more on answering the questions they stated to be important to justify their decision. Furthermore, we clarified any unclear terms with the interviewees if necessary.

Some questions relate to concrete files or methods, which the interviewer selected from the interviewee's system (cf. Sec. III-C). To avoid biases, the interviewees were not allowed to look at any information regarding their system while we prepared 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 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 themes.

TABLE II: 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	[31]	Can you draw a simple architecture of your system?
		$\mathbf{A_{A1}}$: A drawn model *<updated after each other section>*
A_2	[57]	Is a database functionality implemented in your system?
A_3	[54]	Is a user interface implemented in your system?
		$\mathbf{A_{A2,\ A3}}$: ○ Yes: *<file>* ○ No
A_4	[30]	Can you name a file that acts as the main controller of your system?
		$\mathbf{A_{A4}}$: ○ Yes: *<file>* ○ Yes: *<functionality>* ○ No
A_5	[31]	On which other functionalities does the file *<file>* rely?
		$\mathbf{A_{A5}}$: Open text
A_6	[31]	Can you exemplify a file/functionality you implemented using a library?
		$\mathbf{A_{A6}}$: ○ Yes: *<file>* ○ Yes: *<functionality>* ○ No
Section: Meta-Knowledge		
M_1	[38]	When in the project life-cycle has the file *<file>* last been changed?
		$\mathbf{A_{M1}}$: Open text
M_2	[31]	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	[31]	Who is the owner of file *<file>*?
M_6	[31]	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	[31]	What is the intent of the code in the file?
		$\mathbf{A_{C1}}$ *<per file>*: Open text
C_2	[31]	Is there a code smell in the code of the file?
		$\mathbf{A_{C2}}$ *<per file>*: ○ Yes: *<smell>* ○ Yes ○ No
C_3	[54]	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	[31]	Which parameters does the following method need?
C_5	[28]	What type of data is returned by this method?
C_6	[54]	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, 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 from our SLR

C. 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 revealing 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). We selected files and methods ourselves, without the interviewees seeing them. Moreover, we focused on selecting different files and methods in terms of, for instance, their position in the file system, size, last change, and parameters. 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 (for A_i, M_i, and C_i), allowing them to look into their system and re-evaluate what answers had been correct.

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. [66], 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 for similar studies (cf. interview, observational, and think-aloud studies in Tbl. I), and our results yield comparable findings regarding the importance of question (compare Fig. 1c and Fig. 2a). Consequently, we argue that our sample size is reasonable to tackle our research questions.

TABLE III: 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

D. Interviewees

Following recommendations [66], we aimed to include subjects based on differences rather than similarity. To achieve this goal, we invited former or recent collaborators from different countries (e.g., Germany, Sweden, France), who have five to over ten years of programming experiences, are or were active programmers of their system, are working full-time in industry (5) or have industrial experiences from previous positions (4), and are female (3). In Tbl. III, we provide an overview of the respective systems, where the area refers to the system, not the interviewee. Most of the systems had no dedicated documentation. The systems span various domains and programming languages, have been or are still developed for three months to over 10 years, and have between 1 and 6 regular developers. One limitation we can see is that most systems are rather small in terms of size and collaborating developers (even though one system had far more than 50 contributors over the years). *So, while our interviewees represent a diverse sample, we can generalize our observations only for* ***smaller systems****.*

E. Rating Correctness

We rated the correctness of answers by iterating through the corresponding questions 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, imagine that 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 the correctness of the architecture (A_1), which can be subjective. We decided to allow any representation and the interviewees could correct or detail the model during the interview. For rating, we reasoned with the interviewees to understand what they missed and 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?
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 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.

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 comprised corrections at the end.

IV. 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.

A. *RQ1: 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 recall knowledge about their system, but before assessing their correctness.

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 related 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

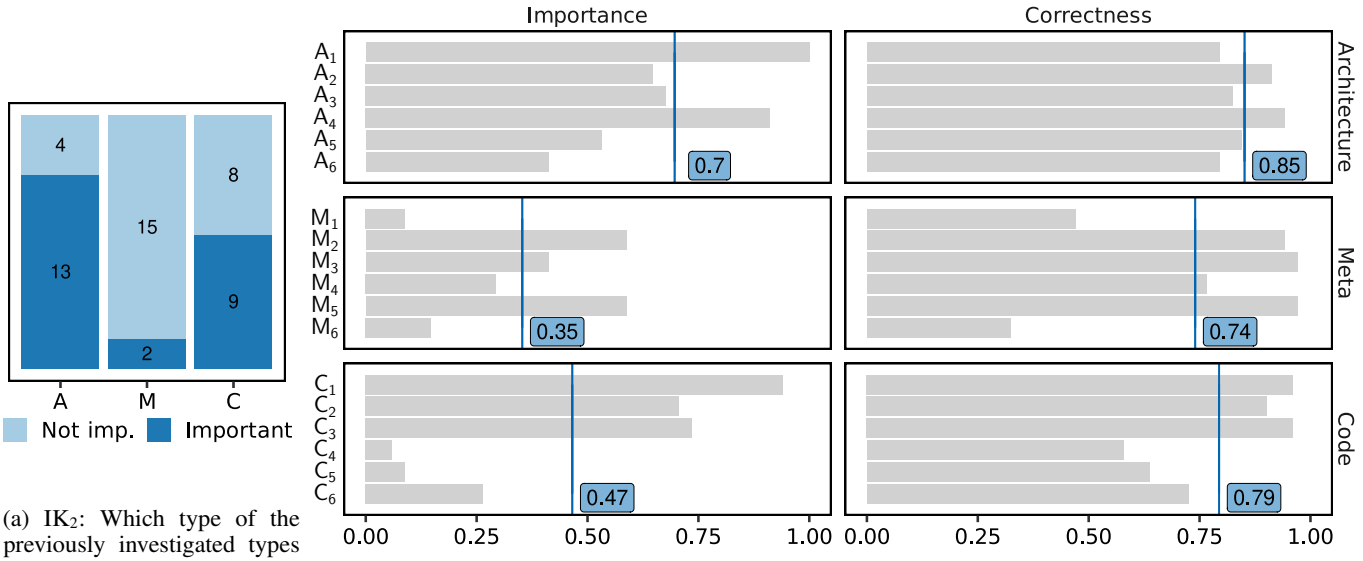
• Intuitively, most interviewees consider the architecture to be an important type of knowledge. • Some interviewees thought about dependencies (i.e., architecture) and intentions (i.e., code) of the system. •

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 Fig. 2a and for IK_3 in Fig. 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. Fig. 1c), rating questions about architecture and code as more important than those on meta knowledge. However, as we were analyzing smaller systems with few developers, we also expected that meta knowledge would be considered less important.

Observation 2

• Most interviewees consider architectural knowledge important to remember. • Few interviewees consider 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



(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 each category and type of knowledge.

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

that in many cases in which interviewees did not consider these two concepts important, the systems did not implement them. Slightly over half of our interviewees considered it important to know which functionalities 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 know 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 individual 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. Particularly, this discrepancy is interesting since meta questions being rated important does not align with our expectations for smaller systems. So, this rating may reflect on larger systems, too—but further studies in larger, collaborative software development are needed.

Observation 4

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

In Fig. 2b, we can see that code questions can be split into two groups regarding the importance to recall them from memory. More than 70 % of our interviewees considered it important to be able to recall 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 for smaller systems:

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

⇒ While meta knowledge is considered less important in general, we found a discrepancy when comparing this to the importance of concrete questions. Arguably, this finding is 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. Fig. 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 architectural 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 can be 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.

To answer **RQ$_1$**, we can see 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, can be encoded in the code, and may be 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.

B. *RQ$_2$: Correctness and Importance*

Next, we compare how important our interviewees' considered it to recall knowledge compared to their correctness.

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. Namely, the questions for the size of a file (M_6) and the last time a file was 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 Fig. 3—also considering the time between the interviewee's last edit and interview in days. Note that we removed five entries

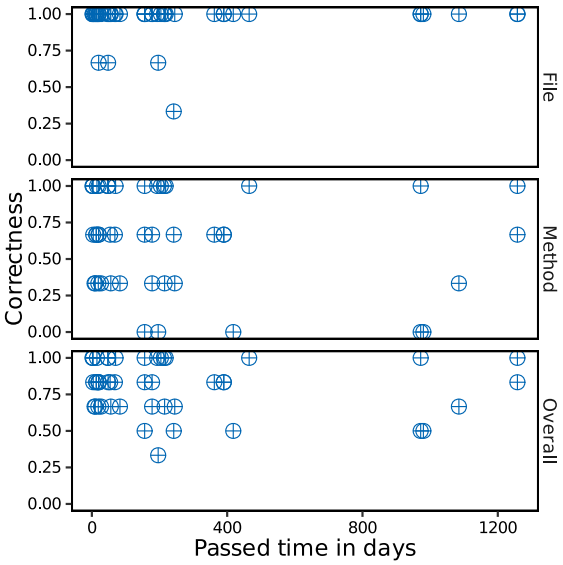


Fig. 3: Comparison of interviewees' correctness on code questions (File: $C_{1–3}$; Method: $C_{4–6}$; Overall: $C_{1–6}$).

(cf. Sec. III-E): 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 R [42], whether this observable difference may be significant, we first tested whether the results are independent of time. We used Kendall's τ because it allows to test non-normal distributions (i.e., correctness) and found 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 a negative tendency of -0.136). This indicates that the difference we can see may depend solely on the type of knowledge. We tested this hypothesis using the Wilcoxon signed-rank test for not normally distributed and paired data to compare the means of both 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 Fig. 2b, we can see that the correctness of the answers resembles the importance of questions. Still, the averages of correct answers are far closer to each other than those of importance. We can also see that only comparably unimportant questions resulted in fewer correct answers. More precisely, none of the questions reported important by more than 50 % of our interviewees received less than 75 % correct answers. To test

whether importance and correctness correlate (e.g., important questions may be harder to check or are kept in mind), we again used Kendall's τ. It reveals a significant, moderately positive correlation between both aspects ($\tau = 0.508$, 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 recalling knowledge they consider important, or forget knowledge they consider unimportant.

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, implying the need for tailored program-comprehension support for different developers (e.g., experts, novices). For instance, our results suggest that particularly code-level program comprehension is important to support, as developers do not recall code details, whereas experts memorize architectures and code intents. For *practice*, our results indicate that developers who are experienced with a system have a good understanding of its architecture, and can potentially guide others.

⇒ In contrast to Krüger et al. [27], 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.

⇒ An important *research* problem that we indicated in Sec. II-B and found again, is the question what the importance of knowledge actually implies? Apparently, developers can recall the knowledge they consider important better. However, this knowledge may simply be important because it is challenging to check or not easy to recover with existing tools. Tao et al. [62] investigated this problem and identified similar issues, but they focused solely on code changes and did not investigate developers' memory.

So, to answer **RQ$_2$**, we find that developers perform quite well in answering questions about their (smaller) 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.

C. *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, and how they derived their self-assessments (IK_4).

Results. In Fig. 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

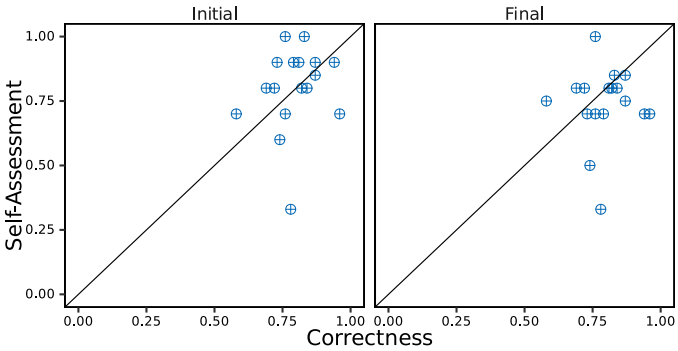


Fig. 4: Correctness compared to overall self-assessment (OS_1).

they summarize the other self-assessments. Interestingly, only one interviewee increased their assessment (+5 %), while eight kept it as it was. On average, our interviewees actually decreased their assessment (-13.75 %). Note that we cannot discuss whether a self-assessment is a perfect prediction of the correctness of answers 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.

Observation 8

• Our interviewees' self-assessment did not change heavily during the interviews. • The initial self-assessment seems closer to the participants' correctness for our questions and smaller systems compared to the final self-assessment. •

Existing research assumes a correlation between self-assessments and a developer's knowledge or experiences, using self-assessments in guidelines and studies [15], [23], [27], [52]. 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 $\tau = 0.176$, final $\tau = 0.032$). However, this outcome is not surprising, as our analysis 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 found neither confirming nor refuting evidence for the assumptions in previous works, indicating the need for more detailed analyses.

Observation 9

• There is no significant correlation (but 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 had for another system they were 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 was perceived less important by the interviewees.

Observation 10

• Our interviewees reflected about various aspects, mainly architecture, effort, and intent, to assess their knowledge. • Some 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 adapted their assessments, most reduced their score, resulting in a more negative perception of their knowledge. 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) recalling 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 system relate to the effort of working on it again, and investigating the impact of different types of knowledge on these efforts.

To answer **RQ$_3$**, we found that developers' self-assessments about their knowledge seems to decline while reflecting about their system. However, we could not identify a correlation between self-assessments and correctness. Further, we identified aspects developers reflect about during self-assessments.

V. Threats to Validity

Internal Validity. Our study was concerned with psychological aspects and other human factors (i.e., developers' memory, knowledge, and opinions). So, there are numerous background factors, such as age, gender, motivation, or a subject's memory performance, that we cannot control. Another threat to the internal validity are the questions we used, which may have not been ideal for our study, may not be ideally ordered, or could be misunderstood. We limited these threats by grounding the question selection in empirical evidence, conducting face-to-face interviews to clarify confusions, testing the interviews, and carefully comparing unavoidable biases against each other.

External Validity Due to the qualitative nature of our study, its small sample size, and the smaller systems we analyzed, the external validity of our results is threatened. We aimed to mitigate such threats by involving diverse developers (e.g., different countries, domains, systems), looking for saturation, and comparing our findings against related work. Still, while the results seem to be reliable for other systems, we emphasize that we can only generalize them for smaller systems.

Another threat to the external validity is that we considered only knowledge in general. However, for instance, the knowledge developers consider important may depend on the task at hand (e.g., fixing a bug versus introducing a new feature). Our results may have been different if we focused on a specific task, and we need to investigate this in future work.

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 may 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], [65], [66], [68] 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.

VI. Conclusion

In this paper, we investigated what knowledge developers consider important to remember, whether they can correctly recall knowledge, and how good their self-assessments are. For this purpose, we reviewed the literature to capture the state of the art on information needs and design an interview survey. Interviewing 17 developers, we found that for smaller systems:

- Developers consider abstract knowledge (e.g., architecture and code intentions) more important to remember.
- Developers are more often correct when recalling knowledge regarding the questions they consider important.
- Developers' self-assessments may be reliable, but their assessments decrease while reflecting about their system.

With our study, we investigated the link between developers' memory and their information needs, extending the existing body of knowledge and providing important insights for research and practice. In future work, we will extend and replicate our study, involving more subjects, other research methods, larger systems, and more detailed analyses of knowledge types.

Acknowledgments Supported by DAAD (IFI fellowship) and DFG (SA 465/49-3). We thank our interviewees for their help.

References

[1] A. Al-Nayeem, K. Ostrowski, S. Pueblas, C. Restif, and S. Zhang, "Information Needs for Validating Evolving Software Systems: An Exploratory Study at Google," in *International Conference on Software Testing, Verification and Validation*, ser. ICST. IEEE, 2017.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix this Bug?" in *International Conference on Software Engineering*, ser. ICSE. ACM, 2006.

[3] G. Avelino, L. Passos, F. Petrillo, and M. T. Valente, "Who Can Maintain This Code? Assessing the Effectiveness of Repository-Mining Techniques for Identifying Software Maintainers," *IEEE Software*, vol. 36, no. 6, 2018.

[4] L. Averell and A. Heathcote, "The Form of the Forgetting Curve and the Fate of Memories," *Journal of Mathematical Psychology*, vol. 55, no. 1, 2011.

[5] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The Impact of Identifier Style on Effort and Comprehension," *Empirical Software Engineering*, vol. 18, no. 2, 2013.

[6] F. O. Bjørnson and T. Dingsøyr, "Knowledge Management in Software Engineering: A Systematic Review of Studied Concepts, Findings and Research Methods Used," *Information and Software Technology*, vol. 50, no. 11, 2008.

[7] J. Börstler and B. Paech, "The Role of Method Chains and Comments in Software Readability and Comprehension – An Experiment," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, 2016.

[8] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain," *Journal of Systems and Software*, vol. 80, no. 4, 2007.

[9] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Frequently Asked Questions in Bug Reports," University of Calgary, Tech. Rep. 2009-924-03, 2009.

[10] C. S. Campbell, P. P. Maglio, A. Cozzi, and B. Dom, "Expertise Identification Using Email Communications," in *International Conference on Information and Knowledge Management*, ser. CIKM. ACM, 2003.

[11] G. Cohen and M. A. Conway, *Memory in the Real World*. Psychology Press, 2007.

[12] B. de Alwis and G. C. Murphy, "Answering Conceptual Queries with Ferret," in *International Conference on Software Engineering*, ser. ICSE. ACM, 2008.

[13] E. Duala-Ekoko and M. P. Robillard, "Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study," in *International Conference on Software Engineering*, ser. ICSE. IEEE, 2012.

[14] A. Erdem, L. Johnson, and S. Marsella, "Task Oriented Software Understanding," in *International Conference on Automated Software Engineering*, ser. ASE. IEEE, 1998.

[15] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring Programming Experience," in *International Conference on Program Comprehension*, ser. ICPC. IEEE, 2012.

[16] R. Feldt, L. Angelis, R. Torkar, and M. Samuelsson, "Links Between the Personalities, Views and Attitudes of Software Engineers," *Information and Software Technology*, vol. 52, no. 6, 2010.

[17] T. Fritz and G. C. Murphy, "Using Information Fragments to Answer the Questions Developers Ask," in *International Conference on Software Engineering*, ser. ICSE. ACM, 2010.

[18] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-Knowledge: Modeling a Developer's Knowledge of Code," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 2, 2014.

[19] J. C. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter Identifier Names Take Longer to Comprehend," *Empirical Software Engineering*, vol. 24, no. 1, 2019.

[20] K. Kang and J. Hahn, "Learning and Forgetting Curves in Software Development: Does Type of Knowledge Matter?" in *International Conference on Information Systems*, ser. ICIS. AIS, 2009.

[21] B. A. Kitchenham, D. Budgen, and P. Brereton, *Evidence-Based Software Engineering and Systematic Eeviews*. CRC, 2015.

[22] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *International Conference on Software Engineering*, ser. ICSE. IEEE, 2007.

[23] A. J. Ko, T. D. Latoza, and M. M. Burnett, "A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants," *Empirical Software Engineering*, vol. 20, no. 1, 2015.

[24] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, 2006.

[25] J. Krüger and T. Berger, "An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE. ACM, 2020.

[26] J. Krüger, C. Lausberger, I. von Nostitz-Wallwitz, G. Saake, and T. Leich, "Search. Review. Repeat? An Empirical Study of Threats to Replicating SLR Searches," *Empirical Software Engineering*, vol. 25, no. 1, 2020.

[27] J. Krüger, J. Wiemann, W. Fenske, G. Saake, and T. Leich, "Do You Remember This Source Code?" in *International Conference on Software Engineering*, ser. ICSE. ACM, 2018.

[28] J. Kubelka, A. Bergel, and R. Robbes, "Asking and Answering Questions During a Programming Change Task in Pharo Language," in *Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU. ACM, 2014.

[29] J. Kubelka, R. Robbes, and A. Bergel, "Live Programming and Software Evolution: Questions During a Programming Change Task," in *International Conference on Program Comprehension*, ser. ICPC. IEEE, 2019.

[30] T. D. LaToza and B. A. Myers, "Developers Ask Reachability Questions," in *International Conference on Software Engineering*, ser. ICSE. ACM, 2010.

[31] ——, "Hard-to-Answer Questions about Code," in *Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU. ACM, 2010.

[32] S. Letovsky, "Cognitive Processes in Program Comprehension," *Journal of Systems and Software*, vol. 7, no. 4, 1987.

[33] S. Lin, W. Hong, D. Wang, and T. Li, "A Survey on Expert Finding Techniques," *Journal of Intelligent Information Systems*, vol. 49, no. 2, 2017.

[34] D. W. McDonald and M. S. Ackerman, "Expertise Recommender: A Flexible Recommendation System and Architecture," in *Conference on Computer Supported Cooperative Work*, ser. CSCW. ACM, 2000.

[35] A. Mockus and J. D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," in *International Conference on Software Engineering*, ser. ICSE. IEEE, 2002.

[36] T. P. Moran, "Anxiety and Working Memory Capacity: A Meta-Analysis and Narrative Review," *Psychological Bulletin*, vol. 142, no. 8, 2016.

[37] S. Nielebock, D. Krolikowski, J. Krüger, T. Leich, and F. Ortmeier, "Commenting Source Ccode: Is it Worth it for Small Programming Tasks?" *Empirical Software Engineering*, vol. 24, no. 3, 2019.

[38] R. Novais, C. Brito, and M. Mendonça, "What Questions Developers Ask During Software Evolution? An Academic Perspective," in *Workshop on Software Visualization, Evolution, and Maintenance*, ser. VEM, 2014.

[39] C. Parnin and S. Rugaber, "Programmer Information Needs after Memory Failure," in *International Conference on Program Comprehension*, ser. ICPC. IEEE, 2012.

[40] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information Needs in Contemporary Code Review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, 2018.

[41] G. Pinto, W. Torres, and F. Castor, "A Study on the Most Popular Questions about Concurrent Programming," in *Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU. ACM, 2015.

[42] R Core Team, *R: A Language and Environment for Statistical Computing*, 2020. [Online]. Available: https://www.R-project.org

[43] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. A. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong, "On-Demand Developer Documentation," in *IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME. IEEE, 2017.

[44] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do Professional Developers Comprehend Software?" in *International Conference on Software Engineering*, ser. ICSE. IEEE, 2012.

[45] I. Rus, M. Lindvall, and S. Sinha, "Knowledge Management in Software Engineering," *IEEE Software*, vol. 19, no. 3, 2002.

[46] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending Studies on Program Comprehension," in *International Conference on Program Comprehension*, ser. ICPC. IEEE, 2017.

[47] D. Schuler and T. Zimmermann, "Mining Usage Expertise from Version Archives," in *International Working Conference on Mining Software Repositories*, ser. MSR. ACM, 2008.

[48] Y. Shakeel, J. Krüger, I. von Nostitz-Wallwitz, C. Lausberger, G. Campero Durand, G. Saake, and T. Leich, "(Automated) Literature Analysis - Threats and Experiences," in *International Workshop on Software Engineering for Science*, ser. SE4Science. ACM, 2018.

[49] K. Y. Sharif, M. English, N. Ali, C. Exton, J. J. Collins, and J. Buckley, "An Empirically-Based Characterization and Quantification of Information Seeking Through Mailing Lists During Open Source Developers' Software Evolution," *Information and Software Technology*, vol. 57, 2015.

[50] V. S. Sharma, R. Mehra, and V. Kaulgud, "What do Developers Want? An Advisor Approach for Developer Priorities," in *International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE. IEEE, 2017.

[51] J. Siegmund, "Program Comprehension: Past, Present, and Future," in *International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER. IEEE, 2016.

[52] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and Modeling Programming Experience," *Empirical Software Engineering*, vol. 19, no. 5, 2014.

[53] J. Siegmund and J. Schumann, "Confounding Parameters on Program Comprehension: A Literature Survey," *Empirical Software Engineering*, vol. 20, no. 4, 2015.

[54] J. Sillito, G. C. Murphy, and K. de Volder, "Questions Programmers Ask During Software Evolution Tasks," in *International Symposium on Foundations of Software Engineering*, ser. FSE. ACM, 2006.

[55] ——, "Asking and Answering Questions During a Programming Change Task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, 2008.

[56] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," in *CASCON First Decade High Impact Papers*, ser. CASCON. IBM, 2010.

[57] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE. ACM, 2015.

[58] W. Stacy and J. MacMillan, "Cognitive Bias in Software Engineering," *Communications of the ACM*, vol. 38, no. 6, 1995.

[59] T. A. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, 1984.

[60] M.-A. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," in *International Workshop on Program Comprehension*, ser. IWPC. IEEE, 2005.

[61] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API Documentation," in *International Conference on Software Engineering*, ser. ICSE. ACM, 2014.

[62] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do Software Engineers Understand Code Changes? An Exploratory Study in Industry," in *International Symposium on the Foundations of Software Engineering*, ser. FSE. ACM, 2012.

[63] R. Tiarks, "What Maintenance Programmers Really do: An Observational Study," in *Workshop on Software Reengineering*, ser. WSR, 2011.

[64] A. von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, vol. 28, no. 8, 1995.

[65] C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering," in *International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE. ACM, 2014.

[66] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[67] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall, "Supporting Developers with Natural Language Queries," in *International Conference on Software Engineering*, ser. ICSE. ACM, 2010.

[68] T. Zimmermann, "Card-Sorting: From Text to Themes," in *Perspectives on Data Science for Software Engineering*. Elsevier, 2016.