To Memorize or to Document: A Survey of Developers' Views on Knowledge Availability

Jacob Krüger[1][0000−0002−0283−248X] and Regina Hebig[2][0000−0002−1459−2081]

[1] Eindhoven University of Technology, The Netherlands; j.kruger@tue.nl
[2] University of Rostock, Germany; regina.hebig@uni-rostock.de

Abstract. When developing, maintaining, or evolving a system, developers need different types of knowledge (e.g., domain, processes, architecture). They may have memorized (but potentially not documented) the knowledge they perceive important, while they need to recover knowledge that they could not memorize. Previous research has focused on knowledge recovery, but not on what knowledge developers consider important to memorize or document. We address this gap by reporting a survey among 37 participants in which we investigated developers' perspectives on different types of knowledge. Our results indicate that the developers consider certain types of knowledge more important than others, particularly with respect to memorizing them—while all of them should be documented, using specific means. Such insights help researchers and practitioners understand developers' knowledge and documentation needs within processes, thereby guiding practices and new techniques.

Keywords: Human Memory · Forgetting · Knowledge · Documentation

1 Introduction

Developing, maintaining, and evolving a software system requires knowledge regarding various of that system's properties, such as its domain, used technologies, architecture, or development processes [2, 9, 17]. Consequently, developers need to learn, memorize, and recover knowledge about a system if needed, which often leads to expensive program comprehension [15, 20]. To facilitate this cognitively challenging activity, researchers and practitioners are concerned with various related concepts, for instance, onboarding practices [22], providing reliable documentation [1], or reverse engineering information [8].

In this context, a particularly important problem is to understand what knowledge developers need during their tasks and how they obtain it. Previously, we [9] have reviewed 14 studies that elicited hundreds of questions developers ask during their work, spanning different areas like the source code, architecture, testing, collaborators, or processes. Despite some of the studies investigating how important or challenging it is for developers to answer certain questions, these studies are typically concerned with very context-specific questions (e.g., "What are the implications of this change?" [14]). Similarly, research has been conducted on understanding how developers memorize and forget knowledge [7, 13, 17].

Surprisingly, little research has aimed to connect the two areas, essentially asking: **What knowledge do developers consider important to memorize or document?** Addressing this question promises important insights for research and practice alike. For instance, knowledge developers consider important to remember may be documented well due to its importance, but based on empirical findings [1,9,16] it seems more likely that developers do not document it properly because they are certain to remember it. Since developers forget or may leave a project, such tacit and undocumented knowledge can cause severe problems, and recovering it is expensive. Regarding documentation, developers may trust or mistrust specific types, for example, because it is not maintained properly. So, there is an important link between what knowledge developers aim to memorize, what they document, and how much they trust their memory or the documentation.

In this paper, we shed light into this link by reporting the results of a questionnaire survey with 37 participants. We asked our participants for five types of knowledge whether they consider that knowledge important (to memorize), how it should be documented, and how they recover it. Our results highlight that developers consider various types of knowledge as important for their work. However, they do not think that all of it should be memorized, and value available information sources to recover knowledge very differently. In detail, we contribute:

- We present the results of our survey to describe what knowledge developers consider important and how they recover it from what sources.
- We discuss our results to help practitioners deal with tacit knowledge in their processes and about their systems, thereby sketching directions for research.
- We publish the anonymous responses to our survey in a persistent repository.[3]

Overall, we contribute to a better understanding of developers' knowledge needs, memory, and documentation preferences—helping practitioners and researchers tackle the connected challenges.

2 Related Work and Motivation

For any task (e.g., fixing a bug) they perform on a system, developers require knowledge regarding, for instance, the goal of the task, the source code they need to change, and how their change may impact the remaining system. We [9] have previously collected studies that identified questions that developers may have during their tasks. Some of these and similar works aim to provide supportive techniques that help developers recover the required knowledge. Particularly, such techniques aim to reverse engineer information directly from a system to provide reliable documentation [1,8]. Other researchers have proposed ways to deal with knowledge-sharing processes, intending to improve the provisioning of information in an organization and decreasing sharing barriers [2].

Still, developers primarily rely on *program comprehension* to recover knowledge they are missing, which is the cognitively challenging activity of understanding what a certain piece of code does and how it relates to the remaining system by

[3] `https://doi.org/10.5281/zenodo.8391861`

reading through the actual code [15, 19, 20]. Developers focus on the code rather than other documentation, because it is the most reliable piece of information about the system: The code exactly specifies what the system does, while any other documentation may be outdated or wrong [1,5,9,16]. While there are various barriers when it comes to sharing and documenting knowledge, it is widely agreed on that documenting a system beyond its source code is helpful and important. Specifically, reliable documentation can help developers obtain or recover knowledge faster, for instance, when they forgot it [7, 10, 13, 17] or are onboarding [4, 6].

While such works are related to and motivate our study, we are focusing on a complementary goal that links these areas: understanding what knowledge developers aim to memorize or document in what forms. As a concrete example, the onboarding of developers must be scoped based on the available documentation. If the developers of a system do not reliably document their system and processes, but trust their knowledge instead, newcomers can only learn from other developers. However, this takes time away from the experts, and thus the additional effort of reliably documenting could have been worth the investment. To research such directions and understand the link between what developers aim to memorize or document, it is first necessary to understand developers' perceptions of these two options. We aim to move into this direction and provide a better understanding of what knowledge developers consider important to memorize or document. So, we complement the related work with novel insights.

3 Design of the Questionnaire

Research Questions. To achieve our goal, we conducted an exploratory survey. Particularly, we designed a questionnaire in which developers rated the importance of different types of knowledge as well as of memorizing or documenting it. Through open questions on how they typically recover each type of knowledge, we aimed to obtain in-depth insights into the connections between knowledge and documentation. To guide our study, we defined four research questions (RQs):

RQ$_1$ *What knowledge do developers consider important?*
RQ$_2$ *What knowledge do they consider important to memorize or document?*
RQ$_3$ *What information sources help recover knowledge?*
RQ$_4$ *How do developers recover knowledge from the sources?*

Answering these RQs contributes to understanding developers' perceived value of memorizing and documenting knowledge. This, in turn, guides future research and practice in designing new techniques, recommendations, as well as studies for managing knowledge in development processes and on systems.

Questionnaire Design. Using an exploratory questionnaire survey allowed us to combine open-ended with closed questions that we could easily distribute to developers. We started the design by analyzing related [7, 17] and our own previous work [9, 11–13] on developers' memory and knowledge needs. At first, each author derived types of knowledge they considered most relevant based on the related work. In three sessions (one hour each), we merged these types into a single classification, derived questions we wanted to ask for our RQs, and defined

what background information we would need about the participants. Then, we re-iterated twice through the questionnaire ourselves, merging and removing questions or changing the answering options. For instance, we redesigned question $<K>_2$ (cf. Tbl. 1) into its final matrix structure—before this step, each entry was an individual question. At this point, we focused on narrowing down the questionnaire to the key questions and types of knowledge, aiming to limit the time developers would need to answer them. We transferred the questionnaire into the SUNET[4] instance hosted by the University of Gothenburg, and conducted test runs ourselves as well as with colleagues. After fixing a few typos and comprehension problems, we ended up with the questions in Tbl. 1.

As we show in Tbl. 1 ("sections on knowledge"), we ended up with five (merged from nine) types of knowledge. We decided to focus on knowledge about a developer's system rather than its domain or technologies to shed light into software development and maintenance processes. The five final types are:

General Code (GC) Knowledge is concerned with the intentions, rationales, and features of the source code. So, this type of knowledge helps developers understand what the code actually does on an abstract level (i.e., the features implemented) and why it has been implemented. Examples: *What is the purpose of this code? Why was this code implemented this way? Why is this code needed?*

Detailed Code (DC) Knowledge is concerned with the code details, such as variables, methods, and other implementation details. So, this type of knowledge is more detailed than the first one and helps developers understand a specific part of the code. Examples: *Where is this method defined? Is this library code? How overloaded are the parameters to this function?*

Quality and Testing (QT) Knowledge is concerned with bugs and design flaws in the code, as well as the methods how to test, debug, and thus quality assure the code (but not tools). Examples: *Is this tested? Is this entity or feature tested? What are hidden (correlated) code issues that may be affecting quality?*

Static and Dynamic Structure (SD) Knowledge is concerned with the general structure of a program, for example, in terms of class diagrams or flow charts, relating to its design and dependencies. Examples: *How does this code interact with libraries? How is this functionality organized into layers? What depends on this code or design decision?*

Collboration (CO) Knowledge is concerned with understanding how the program evolves and by whom. Examples: *When has a file last been changed? Who made a particular change and why? Who is working on similar issues?*

These are the precise definitions we provided in our questionnaire, including the example questions to improve the comprehensibility of each definition.

Questionnaire Structure. We started our questionnaire with a general introduction into our research, the goal of the survey itself, and that it would take approximately 20 min to complete (based on our test runs). Due to missing funding and ethical concerns, we did not use incentives. Instead, we motivated that our survey helps reflect on perceptions and practices, while also guiding research intended to facilitate developers' work. At the end of the welcome page,

[4] `https://www.sunet.se/services/samarbete/enkatverktyg`

Table 1. Questions and answering options in our questionnaire.

id	questions & answering options
section on information sources (IS)	
IS_1	How valuable do you consider the following information sources? Likert scale <○ not valuable at all ○ not valuable ○ neutral ○ valuable ○ very valuable> `for each` • *own knowledge* • *knowledge of collaborators* • *source code* • *documentation* • *version control system* • *analysis tools (e.g., debuggers)* free text `for` • *Are we missing any information source you consider valuable or very valuable?*
IS_2	How much do you agree with the following statement: *It is important to remember where to search for necessary information.* Likert scale <○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree>
sections on knowledge (<K>), one for each: general code (GC); detailed code (DC); quality and testing (QT); static and dynamic structure (SD); collaboration (CO)	
$<K>_1$	How much do you agree with the following statements: single-choice selection <○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree> `for each` • *It is important to know about <K> knowledge out of memory.* • *It is important to have <K> knowledge available in some other form, e.g. within source code, via supporting tools, or as documentation.*
$<K>_2$	According to your preference, information should be available in the following form: single-choice selection <○ yes ○ no> `for each` • *source code (e.g. code logic or identifier names)* • *additional information in source code (e.g. comments or annotation)* • *documentation (e.g. manuals or models)* • *version control system (e.g. commit)* free text `for` • *other*
$<K>_3$	How do you normally familiarize yourself with <K> knowledge regarding one of your programs? free text
section on importance of knowledge	
IK_1	How important do you consider the different types of knowledge compared to each other? ranking from $<1^{st}$ – most important> to $<5^{th}$ – least important> `for` • *general code* • *detailed code* • *quality and testing* • *static and dynamic structure* • *collaboration*

we informed each participant that their participation is voluntarily and asked for their consent. We noted that they could withdraw at any point, that all responses were anonymous (except what they revealed themselves and B_{12}), and that the data would be used as well as published (in anonymous form) for research.

In the first actual section (IS_i in Tbl. 1), we asked two questions. First (IS_1), we elicited how valuable each participant perceives common information sources, also providing an option to add important ones we did not list (**RQ$_3$**). Second (IS_2), we asked how much each developer agrees that it is important to know where to find relevant knowledge (**RQ$_2$**, **RQ$_4$**). We asked these questions first to avoid biases by participants reflecting on their knowledge and processes in the following sections, aiming to elicit their unbiased opinions first. For both questions, we used Likert scales as an intuitive means to indicate preferences. We allowed participants to select a neutral state in case they do not have a strong opinion about a statement or are not too familiar with the specific topic.

Then, we defined one section for each knowledge type ($<K>_i$ in Tbl. 1), asking the exact same three questions for each. In this paper, we replace <K> in the label of the questions with the abbreviation of the respective type of knowledge (i.e., GC_1 refers to $<K>_1$ for *general code knowledge*). Through the first question ($<K>_1$), we compared the developers' perceptions regarding knowing or documenting knowledge (**RQ$_2$**). Then, we asked ($<K>_2$) how the respective knowledge should be available (**RQ$_3$**) and ($<K>_3$) how the developer typically recovers their knowledge (**RQ$_4$**). We again used Likert scales and single-

choice selections to identify participants' preferences in the first two questions. For the last question, we used free text to allow for detailed descriptions of activities.

After a developer iterated through each type of knowledge, we asked (IK_1) them in the last section to rank the types according to importance ($\mathbf{RQ_1}$). We put this question last to allow each participant to understand the different types first and to reflect on how they build on it during their daily work. In this case, we anticipated that this comprehension is needed and that the reflection on the own practices does not induce biases, but rather improves the trust in the rankings.

We asked 13 questions on the participants' background (not in Tbl. 1, but in our repository[3]), building on guidelines for empirical studies [21] and recommendations by our universities for designing inclusive questions. Specifically, we asked for the country participants work in (B_1), their programming experiences ($B_{2–4}$), the domains (B_5) and example projects (B_6) they have worked with, their mostly used programming languages (B_7), typical team (B_8) and system sizes (B_9), as well as gender (B_{10}) and age (B_{11}). Finally, we asked whether the participants wanted to receive the results of the survey (B_{12}) and for additional comments (B_{13}).

Data Handling. The survey responses were automatically stored on internal servers of the University of Gothenburg to ensure data security. We used a private repository to share the data between both authors. For our analysis, we used only complete responses and discarded dropouts, meaning that the participants answered each question except for background questions they preferred not to answer. However, there is one exception: One participant answered all questions, but put only a single character for all versions of $<K>_3$. We decided to include this otherwise complete survey.

Inviting Developers. We aimed to target developers focusing on different aspects of software engineering, such as programming, testing, or architecting, and with varying backgrounds. It is challenging to define the size of this target population and design an appropriate sampling, due to missing information. Moreover, it is problematic to contact a larger number of potential candidates without spamming them or causing other ethics concerns [3]. For these reasons, we designed the following strategy to contact potential participants.

First, each author listed (open-source) projects that they knew and perceived as having a strong connection to research (intending to increase the response rate), such as the Linux Kernel, Mozilla, Debian, Apache, Rust, or Eclipse. We merged these lists and investigated each project's websites to identify details on research collaborations or contact information for such inquiries. Then, we drafted individual mails motivating why our research may be interesting for them and asking whether there would be any option to invite developers of the project to our questionnaire. In some cases, we received no response (we sent no follow-up mails) or were informed that this would not be possible. In a few cases, we were informed about alternative means (e.g., blog postings), which we tried but were mostly rejected. Fortunately, several project managers were positive about our survey and sent our invitation to developers or mailing lists they deemed interested. Using this process, we aimed to get responses by experienced developers of established systems, while not spamming uninterested developers.

To expand the pool of candidates, we used two more channels to distribute our survey. First, we posted a short invitation on feasible discussion platforms for software developers, such as DanniWeb and Reddit. Second, we submitted our survey to professional participant recruitment platforms, for instance, SurveyCirlce and clickworker. We identified the respective platforms through an unstructured search. In our questionnaire, we did not ask participants how they got into contact with it, and thus we cannot specify how many participants each channel resulted in. One of our participants commented to have found it on DanniWeb, while the participant recruitment platforms indicated that we received only one response—likely due to its members lacking the required background. So, these two channels did unfortunately not really work out.

Participants. At least 83 individuals started our survey to the point that they agreed to the consent statement. Of these, 37 (44.58 %) provided answers to all questions, and are thus included in our analysis. These 37 participants worked in different countries (B_1), specifically Germany (16), Sweden (4), UK (4), USA (2), Switzerland (2), Poland, Russia, Norway, Austria, Mexico, Turkey, Ireland, Brazil, and The Netherlands. Their programming experiences varied: None of the participants had less than a year of general programming experience (B_2), while eight had 1–5, four 6–10, and 25 more then ten years of experience. Regarding industry (B_3), five participants had less than a year of experience, while 12 of them had 1–5, nine 6–10, and 11 more than 10 years of experience. The industrial experiences spanned various domains (B_5), such as transportation, healthcare, robotics, finance, and telecommunications. Nine participants indicated less than a year of experience with open-source systems (B_4; 1–5: 8; 6–10: 6; >10: 14), for which mostly Debian (13 times), but also Linux Kernel, Eclipse, Mozilla, and Apache projects were mentioned (B_6). Consequently, the participants also worked with various programming languages (B_7), such as C/C++ (20), Python (20), Java (15), JavaScript (11), and many others (e.g., Ruby, Rust, Perl, Bash). Similarly, the sizes of the teams (B_8) they typically worked in (1: 3; 2–4: 13; 5–8: 16; 9–15: 2; >15: 3) and of the systems (B_9) varied (<10,000 LoC: 10; <100,000 LoC: 10; <1,000,000 LoC: 12; >1,000,000 LoC: 5). Regarding their gender (B_{10}), 25 participants indicated to be male, one to be female, and all others preferred not to tell. Finally, (B_{11}) 17 participants were 36–45, four 18–25, three 46–55, and two 66–75 years old (the rest preferred not to tell). Overall, this sample of participants spans a variety of characteristics and backgrounds (except for gender, a typical problem in software engineering), representing different experiences in developing software. So, we argue that our participants are representative of typical software developers.

Data Analysis. We downloaded the data for all 37 responses as a spreadsheet. Using R [18], we derived summarizing statistics and plots (e.g., Fig. 3) for all questions that had fixed answering options (i.e., Likert scales, single-choice selection, ranking) to analyze the distribution of our data. For the free-text answers and "other" options, we used methods for qualitative document analyses. Specifically, we employed open coding to identify important information related to developers' activities, documentation, and knowledge. Then, we employed open-card sorting [23] to derive higher-order themes from our codes. We used the statistics,

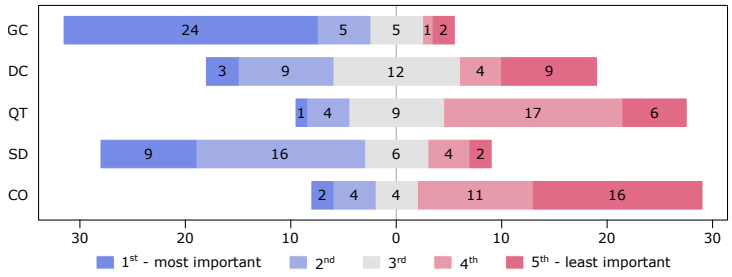

Fig. 1. IK_1: Ranking the types of knowledge.

plots, and insights from the "other" options to answer $\mathbf{RQ_1}$, $\mathbf{RQ_2}$, and $\mathbf{RQ_3}$. By reflecting on the statistics, plots, and free-text analysis, we answered $\mathbf{RQ_4}$.

4 Results and Discussion

Next, we report and discuss the results we obtained for each RQ.

4.1 $\mathbf{RQ_1}$: Important Knowledge

Results. In Fig. 1, we summarize our participants' responses to IK_1, how important they would rank each type of knowledge. As we can see, the ranking of the individual types is rather clear. GC knowledge is perceived as most important with a majority of 24 participants ranking it in first place. The SD knowledge follows in second place with nine participants ranking it first and 16 ranking it second. For DC knowledge, the rankings are tending towards third place (12 responses), and it was ranked almost equally as more (12) and less important (13). QT knowledge was most often put into fourth place (17), and rarely into one of the first two places (5). Finally, the least important is the CO knowledge, with 16 and 11 participants ranking it last and fourth, respectively.
Discussion. From our data, we can derive that our participants perceive more abstract knowledge (GC, SD) about their system as more important. This finding is in line with our other works [9,10], underpinning such findings and also improving our trust in the results of our survey. For researchers as well as practitioners, this implies that techniques and supportive means for documenting, maintaining, and recovering more structural or conceptual knowledge about a system is important to support developers, for instance, during their onboarding in a project.

Interestingly, QT and CO knowledge are perceived least important. The former may rank lower because our participants perceive it as the task of others and not themselves, to quote one of them: *"QA's job, not mine."* However, considering that many participants mentioned test cases as artifacts they look into when recovering knowledge (cf. Sec. 4.4), it seems unlikely that this is the primary cause. Similarly, while CO knowledge is ranked last, collaborators and version-control data are regularly mentioned as means for recovering knowledge. So, we consider it more likely for both types of knowledge that they are simply perceived as less important by our participants compared to the other types. This aligns with our results for $\mathbf{RQ_2}$

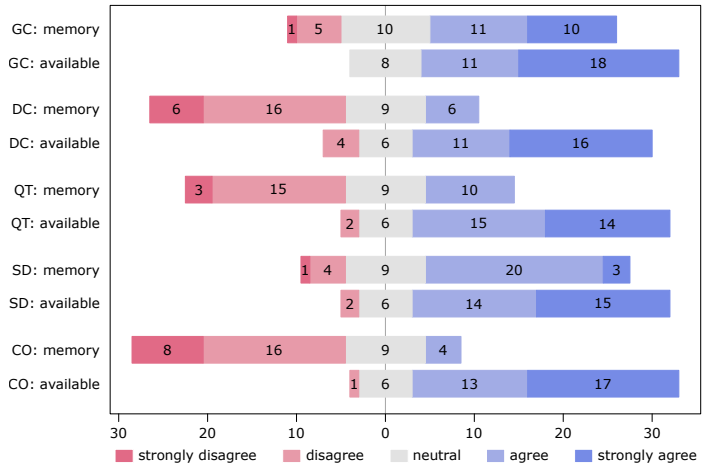

Fig. 2. <K>$_1$: How much do you agree that it is important to know <K> out of **memory** and to have <K> **available** in some other form?

(cf. Sec. 4.2): On average, all types of knowledge receive strong agreement that respective information should be available, while the more important ones should also be memorized. This also matches with DC knowledge ranking third. It can be recovered during program comprehension and is mostly relevant for a specific task, during which it can be recovered and does not need to be memorized. This implies that, while perceived as less important, supporting developers with relevant information about QT, CO, and DC knowledge can still facilitate their tasks.

RQ$_1$: Important Knowledge

Developers perceive more abstract knowledge (GC, SD) as more important.

4.2 RQ$_2$: Memorizing and Documenting

Results. In Fig. 2, we display the agreement of our participants with question <K>$_1$ regarding to what extent they consider it important to memorize knowledge or to have it available. We can see that a majority perceives it important to have information available across all types of knowledge. This is further underpinned by the answers to IS$_2$, with 20 participants strongly agreeing and 13 agreeing that it is important to remember where to search for information. Only three responses to IS$_2$ are neutral and one disagrees with the statement.

Contrary to the availability, we can see a clear difference in Fig. 2 when considering memory, which aligns to our previous results for **RQ$_1$**: The two more abstract types of GC and SD knowledge are rated more important to memorize. In contrast, the other three types are less often considered important to memorize. More specifically, only four, six, and ten participants consider it somewhat important to memorize CO, DC, and QT knowledge, respectively.

Discussion. Developers consider it more important to memorize abstract (GC, SD) knowledge about their system, which helps them obtain an overarching understanding of a system. Based on this knowledge, they would be capable of

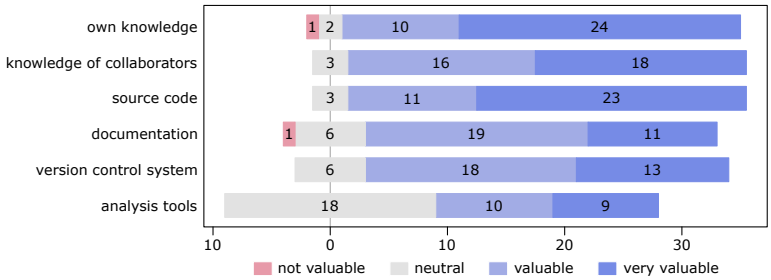

Fig. 3. IS_1: How valuable are these information sources?

narrowing down what parts of a system are relevant for a task. In our previous research, we have hypothesized that developers may focus on remembering more abstract knowledge and recover details during program comprehension [9, 13]. Our new results provide supportive evidence that this is indeed the case. For researchers, this insight can help design techniques for reverse engineering information and guiding system explorations. A particular challenge for researchers and practitioners is to document tacit knowledge of developers to make it available to others and identify the experts for a specific piece of a system.

It is not surprising that a majority of our participants perceives it important to have information available across all types of knowledge. We argue that this reflects that any type of knowledge can become important for a task or process. For any task in which knowledge may not be available and must be recovered, additional, reliable information about it can facilitate the developers' work. Also, we argue that the responses show that important knowledge developers aim to memorize should still be documented in case it is forgotten, and that detailed knowledge that is not considered important to memorize is more challenging to recover. In either case, having information available is helpful for the developers.

RQ_2: Memorizing and Documenting

Developers consider it more important to memorize abstract knowledge (GC, SD) about a system, but prefer to have information for any type available.

4.3 RQ_3: Information Sources

Results. In Fig. 3, we display how valuable our participants consider different information sources. We can see that especially own knowledge, knowledge of collaborators (despite the low ranking of CO knowledge), and source code are perceived as highly valuable, with each receiving 34 valuable or highly valuable scores. Documentation and the version control system are also perceived valuable, but to a slightly lesser extent. Interestingly, 18 of our participants consider analysis tools neutrally, while 19 think these are (very) valuable information sources.

In addition to the information sources we listed, our participants mentioned several others as important. Via our open coding (codes and multiple occurrences in parentheses) and card sorting (bold label), we derived seven categories:

Online resources (discussion forums: 3; Q&A sites: 2; web search: 2; blog posts; video tutorials) have been mentioned repeatedly. Typically, such resources

Table 2. $<K>_2$: In what form should information be available? We use colors to highlight cases in which at least 80 % of the participants agreed (i.e., at least 30).

<K>	code		add.		docu.		vcs		others (open coding)
	yes	no	yes	no	yes	no	yes	no	
GC	30	7	25	12	25	12	18	19	collaborator, homepage, architecture diagrams, readme, tickets
DC	34	3	23	14	9	28	16	21	test cases, coding tools, tickets
QT	21	16	15	22	19	18	11	26	test cases, tools, test reports
SD	28	9	23	14	32	5	4	33	collaborator, tools
CO	4	33	1	36	9	28	37	0	tickets, collaborator, communication, tools

provide examples or discuss concrete problems of a system. This is valuable information, but does only exist for more prominent systems.

Stakeholders (product owner; users) can provide valuable information, particularly on the intended use of a system and its requirements.

Test cases (test case: 2) have been mentioned as a specific piece of source code with particular value. Specifically, test cases (ideally) serve as descriptions of the code's intended use, and provide examples of how to use the code.

Specifications (use case, standard, formal specification) are helpful means to understand what a piece of software has to fulfill.

Packages (dependencies, library authors) are structural elements as well as reusable components within a system. Their dependencies can help understand the structure of a system, while their authors are experts to ask for help.

Issues (tickets, issue histories) provide insights into what requirements a system should fulfill, and their history reveals how this changed over time.

Monitoring data (monitoring data) can provide insights into the actual behavior of the system at runtime.

Note that we consider some of the codes to overlap with the sources we listed. For instance, we would expect specifications to be part of or being referenced in documentation and stakeholders involved in a system to be considered collaborators.

In Tbl. 2, we summarize our participants' opinions on how information for each type of knowledge should be available. We highlight large majorities (i.e., at least 80 % / 30 responses) with green (agreement) and red (disagreement) background. In the last column, we list the additional information sources we derived via our open coding. As we can see, most combinations of type of knowledge and information source are somewhat balanced, with several developers considering it relevant while others do not. In contrast, a majority of our participants agrees that GC and DC knowledge should be documented directly in the source *code*, for instance, following clean-code principles. A majority of our participants does not agree that CO knowledge should be available in the source code or *add*itional code elements (e.g., author tag in comments), but all of them agree that the version-control system (*vcs*) should be used for that. Regarding SD knowledge, particularly *docu*mentation is perceived as a relevant information source whereas the version-control system is not. Note that these ratings align to the results of our open coding for $\mathbf{RQ_4}$ in Sec. 4.4 (cf. Tbl. 3).

Discussion. The differences in how important our participants' perceive a certain information source (Fig. 3) for a type of knowledge (Tbl. 2) is likely caused by their individual experiences. Consequently, we expected the outcome that many different sources exist and will be mentioned, and that our abstracted sources

will mostly receive mixed ratings. However, the strong tendencies in Tbl. 2 in which at least 80 % of our participants agree, highlight clear preferences.

It is not surprising that developers value their own and collaborators' knowledge as well as source code the most. The former two build the cognitive understanding of the system, while the latter is exactly representing the system's behavior. Similarly, we are not surprised that the developers see value in the version control system, since it documents all changes automatically and provides additional information by collaborators (e.g., commit messages, issues). In contrast, we are surprised by how valuable our participants perceive documentation, since it is often considered unreliable and outdated (cf. Sec. 4.4). Also, the even split between neutral and (very) valuable for analysis tools is interesting, which we take as an indicator that such tools are sparsely used by our participants. Reflecting on the perceived value of information sources, we argue that further research is needed to understand the reliability of developers' knowledge.

Considering Tbl. 2, it is intuitive that our participants agree that particularly GC and DC knowledge should be available directly in the source code. Interestingly, and aligning to our previous discussion, documentation is perceived particularly important to make SD structure knowledge available. Arguably, it is more challenging to recover such structural information from the source code, and documentation can severely reduce the effort. We are a bit surprised that almost all participants agree that neither source code nor code additions (e.g., annotations, comments) should be used to make CO knowledge available, since author and contributor tags are often used in code comments. Finally, the version control system itself is unsurprisingly not considered for making SD knowledge available, but all participants agree that it is the primary source for CO knowledge. Considering the other forms in which knowledge should be made available, these align with our previous categorization, including collaborators, test cases, tools, or tickets. Overall, we can see from these results that our participants seem to perceive different strength and weaknesses regarding individual information sources. For researchers and practitioners, it is important to reflect on these perceptions to design useful techniques and management strategies.

RQ$_3$: Information Sources

Developers' knowledge and source code are perceived as most valuable source, but other sources (e.g., documentation) should not be neglected.

4.4 RQ$_4$: Recovering Knowledge

Results. For <K>$_3$, our participants provided highly varying levels of details. Some explained in great detail the processes of how they recover what type of knowledge. In a few cases, they left individual descriptions empty or listed only a few information sources they typically look at. To answer **RQ$_4$**, we read the descriptions and coded the mentioned information sources, which we summarize in Tbl. 3. We can see how often each (card sorted) source is mentioned for each type of knowledge, and the sum of all mentions in the last column. In the following, we report the participants' descriptions and Tbl. 3 in more detail.

Table 3. Codes we extracted from the responses to <K>$_3$.

	# occurrences for					
code	**GC**	**DC**	**QT**	**SD**	**CO**	$\sum$
code	23	27	10	18	—	78
documentation	20	9	5	19	4	57
vcs	4	8	1	1	34	48
colleagues	10	7	2	8	9	36
execution	6	5	4	7	—	22
test cases	1	1	19	1	—	22
tools	—	6	7	7	2	22
comments	9	7	3	2	—	21
test reports	—	—	4	—	—	4
search engine	2	1	—	—	—	3
directory structure	2	—	—	2	—	2
change logs	1	—	—	—	1	2
examples	1	—	—	—	—	1
bug injection	—	—	1	1	—	1
logging messages	—	—	—	1	—	1

For **GC knowledge**, most of our participants analyze the source code (23) and documentation (20)—with some explicitly mentioning comments in the code (9), too. Interestingly, most participants indicate to actually start with looking for documentation (e.g., on APIs, readme files) and only afterwards they (may) look at the source code. Investigating the source code typically means that the developers move from a higher level (e.g., classes, header files) down to code details if necessary. Specifically, one participant mentions that *"the high level should be explained in the documentation,"* and another one that the documentation should be the right point to start: *"I start with the documentation looking for overall information like architecture, design patterns etc. (which is usually missing, outdated or incorrect)."* Due to documentation being apparently often unreliable, our participants typically move towards analyzing the source code, but also ask colleagues (10) and execute the system (6), for instance, using debuggers. However, while *"code is always more accurate and up to date than docs, [it is] usually harder to get an overview from [it],"* which is why two of our participants consider *"the code [only] if there is nothing else."* There are some other information sources that developers use when recovering GC knowledge, including test cases, online search engines, directory structures, change logs, or examples. Some developers investigate the version control system to understand why and how code evolved, for instance, by reading commit messages.

For **DC knowledge**, it is not surprising that this type is almost exclusively recovered from the source code itself (27). Specifically, aligning to existing research and the responses above, one participant stated that *"the code is the ground truth, everything else is outdated by definition."* Consequently, it is logical that none of the other information sources is considered particularly often when investigating details of the source code. However, we found two interesting strategies for recovering DC knowledge in those. First, one participant stated that pair sessions are helpful, arguably because the two developers involved explain the code to each other to improve their comprehension. Second, another participant mentions to actually execute test cases to see how the source code behaves.

For **QT knowledge**, we can see that the source code receives less attention (10). Instead, our participants explain to focus on reading and running test cases

(19) and looking at (historical) test reports (4), which is not surprising. Note that we list test cases despite the overlap with code and execution because these have been explicitly mentioned various times. Some other information sources are used to enrich knowledge recovery, but none sticks out in terms of occurrences—even though some a very interesting. For instance, one participant stated to essentially inject bugs into the system to actually understand the quality of the tests (i.e., mutation testing). Some others mention to investigate particularly tools, scripts, and test cases in continuous integration pipelines, bug trackers, and documented testing instructions. Moreover, one participant highlights that *"[...] testing knowledge often needs a synchronization between developers, testers and the business. This knowledge should be shared somewhere [...]."* We consider this a very important reflection that puts the statement we refer to in Sec. 4.1 into context: Even if it is not the responsibility of the developers, testing and quality assuring is a collaborative effort and to facilitate the tasks of everyone involved, a place to share, manage, and maintain knowledge is essential.

For **SD knowledge**, our participants primarily investigate documentation (19) and match it to the source code (18). This aligns with our previous insights for $\mathbf{RQ_3}$ that documentation is perceived particularly helpful to document a general architecture and overview of the system, for instance, using diagrams. One developers states that *"this is either part of the project documentation and/or usually presented as part of the project onboarding."* Again, some other information sources are used somewhat, such as asking colleagues, executing the system, or analysis tools (especially dependency analyses). Again, one participant each stated to use bug injections and to check logs to understand the program flow.

For **CO knowledge**, our investigation into how our participants' recover it clearly shows that they are investigating almost exclusively the version control system (34). Particularly, one participant described: *"Whenever I'm curious about a certain code change (e.g. it looks wrong), I almost always use git blame to see why a certain line was changed."* In fact, we found that typical version control systems seem to provide all means needed to recover CO knowledge, such as issue trackers and commit logs. Some other information sources, such as asking colleagues (9) or consulting documentation (4) and change logs (1), seem to rather be used to gain a general understanding of collaboration practices: *"[...] Almost all interesting result[s] of such interaction will be recorded in the git repository graph as contributed work and can be easily queried, but hints about non-documented design choices can still sometimes be learned from studying the code review interaction, as well as learning about what the project maintainers' care about in terms of code quality and architecture."* So, while a version control system is the most relevant source, it still seems reasonable to extend it with further documentation.

Discussion. Our participants indicated different processes and preferred information sources for recovering certain types of knowledge. The mapping between source and consequent knowledge is intuitive, but we identified some interesting practices. In fact, we believe that in most cases the full potential for documenting and recovering knowledge is not exploited, yet. Also, how to combine the different information sources to recover a reliable knowledge base remains an open issue.

So, our results imply various important research directions to facilitate developers' tasks by providing new techniques—or by improving the adoption of existing ones.

Again, it is clear that source code is the most reliable information source. Still, other sources are used extensively across various (e.g., documentation) or for specific (i.e., test cases) types of knowledge. Seeing how many developers rely on documentation and consider it important (**RQ$_3$**), we argue that the problem of outdated documentation may rather be a self-fulfilling prophecy today. So, it is an important challenge to improve the management and maintenance of information sources; and to convince developers to engage with these. Finally, we identified interesting practices for recovering knowledge that are not explored in research (e.g., bug injection) or lack practical value at the moment (e.g., tools).

RQ$_4$: Recovering Knowledge

While source code represents the ground truth, developers actually perceive many other information sources as valuable for recovering knowledge.

4.5 Threats to Validity

Our questionnaire design (e.g., wording, length) may have caused misunderstandings, not motivated participation enough, or biased participants. To mitigate such threats, we discussed the order and phrasing of questions; and conducted test runs with colleagues to identify and resolve confusions. Second, our sample of participants may have introduced bias, for instance, because they were more connected to open-source systems or because we did not perform demographic analyses. Still, our participants' backgrounds varied, they have also industrial experiences, and their answers mostly hint in the same directions—which is why we argue that this threat is minimal. Finally, our data analysis (e.g., open coding) may have introduced bias due to our interpretation. This is an inherent threat, but we contribute our data to allow others to replicate and validate our study.[3]

5 Conclusion

In this paper, we reported a questionnaire survey with 37 developers regarding the availability, memorization, and documentation of knowledge. Overall, we learned:

- Developers consider abstract knowledge the most important (Sec. 4.1).
- Developers consider it important to have all types of knowledge available, and aim to memorize abstract knowledge (Sec. 4.2).
- Developers consider their own knowledge and source code as the most important information sources, but other sources are also valuable (Sec. 4.3).
- Developers use various information sources as complements; particularly documentation even though its quality is typically poor (Sec. 4.4).

Our findings link different research areas and provide empirical insights that guide future work, for instance, on recording, maintaining, and sharing knowledge.

References

1. Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., Shepherd, D.C.: Software Documentation: The Practitioners' Perspective. In: ICSE. ACM (2020)

2. Anquetil, N., de Oliveira, K.M., de Sousa, K.D., Batista Dias, M.G.: Software Maintenance Seen as a Knowledge Management Issue. Inf Softw Technol (2007)
3. Baltes, S., Diehl, S.: Worse Than Spam: Issues In Sampling Software Developers. In: ESEM. ACM (2016)
4. Dominic, J., Ritter, C., Rodeghero, P.: Onboarding Bot for Newcomers to Software Engineering. In: ICSSP. ACM (2020)
5. Fluri, B., Würsch, M., Gall, H.C.: Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In: WCRE. IEEE (2007)
6. Ju, A., Sajnani, H., Kelly, S., Herzig, K.: A Case Study of Onboarding in Software Teams: Tasks and Strategies. In: ICSE. IEEE (2021)
7. Kang, K., Hahn, J.: Learning and Forgetting Curves in Software Development: Does Type of Knowledge Matter? In: ICIS. AIS (2009)
8. Koschke, R.: Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In: ISSSE. Springer (2009)
9. Krüger, J., Hebig, R.: What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In: ICSME. IEEE (2020)
10. Krüger, J., Hebig, R.: What Data Scientists (Care to) Recall. In: PROFES. Springer (2023)
11. Krüger, J., Mukelabai, M., Gu, W., Shen, H., Hebig, R., Berger, T.: Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. J Syst Softw (2019)
12. Krüger, J., Nielebock, S., Heumüller, R.: How Can I Contribute? A Qualitative Analysis of Community Websites of 25 Unix-Like Distributions. In: EASE. ACM (2020)
13. Krüger, J., Wiemann, J., Fenske, W., Saake, G., Leich, T.: Do You Remember This Source Code? In: ICSE. ACM (2018)
14. LaToza, T.D., Myers, B.A.: Developers Ask Reachability Questions. In: ICSE. ACM (2010)
15. von Mayrhauser, A., Vans, A.M.: Program Comprehension During Software Maintenance and Evolution. Computer (1995)
16. Nielebock, S., Krolikowski, D., Krüger, J., Leich, T., Ortmeier, F.: Commenting Source Code: Is It Worth It for Small Programming Tasks? Empir Softw Eng (2019)
17. Parnin, C., Rugaber, S.: Programmer Information Needs after Memory Failure. In: ICPC. IEEE (2012)
18. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing (2023), `https://www.R-project.org`
19. Roehm, T., Tiarks, R., Koschke, R., Maalej, W.: How do Professional Developers Comprehend Software? In: ICSE. IEEE (2012)
20. Schröter, I., Krüger, J., Siegmund, J., Leich, T.: Comprehending Studies on Program Comprehension. In: ICPC. IEEE (2017)
21. Siegmund, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring and Modeling Programming Experience. Empir Softw Eng (2014)
22. Steinmacher, I.F., Graciotto Silva, M.A., Gerosa, M.A., Redmiles, D.F.: A Systematic Literature Review on the Barriers Faced by Newcomers to Open Source Software Projects. Inf Softw Technol (2015)
23. Zimmermann, T.: Card-Sorting: From Text to Themes. In: Perspectives on Data Science for Software Engineering. Elsevier (2016)