Non-Knowledge as a New Lens on Software Engineering

Jacob Krüger
j.kruger@tue.nl
Eindhoven University of Technology
Eindhoven, The Netherlands
Harz University of Applied Sciences
Wernigerode, Germany

Xenia Marlene Zerweck
Sol Martinez Demarco
Alena Bleicher
Thomas Leich
smartinezdemarco@hs-harz.de
ableicher@hs-harz.de
tleich@hs-harz.de
Harz University of Applied Sciences
Wernigerode, Germany

Abstract

Software engineering is a knowledge-intensive process. Consequently, researchers typically understand any lack of knowledge as a problem that must be mitigated by improving, for instance, program comprehension, reverse engineering, community collaboration, or documentation. However, a lack of knowledge may not always be a problem. In fact, research in the field of ignorance studies has highlighted the diversity of ignorance phenomena and emphasized their relevance in social interaction. We argue that focusing on *non-knowledge* as one of these phenomena can also be useful for software-engineering research. In this paper, we develop and substantiate this claim by providing a brief overview on the (social scientific) research on ignorance and by proposing a working definition of non-knowledge for software-engineering research. Then, we sketch how the perspective of non-knowledge as a social phenomenon can benefit future research within software engineering and propose three concrete directions to investigate. We envision that non-knowledge contributes a new lens to manage the complexity of intellectual capital and knowledge in modern software engineering. With this paper, we hope to motivate and guide future software-engineering research in this direction.

CCS Concepts

• **Software and its engineering**;

Keywords

Knowledge, Non-Knowledge, Ignorance, Cognition

ACM Reference Format:
Jacob Krüger, Xenia Marlene Zerweck, Sol Martinez Demarco, Alena Bleicher, and Thomas Leich. 2025. Non-Knowledge as a New Lens on Software Engineering. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728503

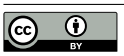
1 Introduction

Software and its engineering are becoming increasingly complex, involving large social communities of developers and users [4, 14], dependencies between various software and hardware systems [6, 9, 16], or different practices and routines [1, 15, 37]. Thus, a primary goal of software engineering as a socio-technical and knowledge-intensive process is to support humans in managing this increasing complexity [32, 33, 38]. A particular challenge in this regard is to deal with knowledge gaps, which are inherent in software development and complex software systems.

Cross-cutting in nature, knowledge about software and its engineering is difficult to document, as it is highly personal in nature, related to experiences and skills of individuals (e.g., assumptions, intentions), and is often scattered across artifacts and tools (e.g., issue trackers, databases, social-coding platforms), which may also have access constraints [5, 18–20, 33]. Also, even documented knowledge is often not maintained if it does not immediately benefit the developers involved (e.g., code comments [8, 25]). Thus, much software-engineering knowledge remains tacit in practice, and can easily be forgotten [19, 21, 27] or become outdated. In turn, software engineers constantly face knowledge gaps in their daily work.

Research in software engineering typically understands knowledge gaps as a problem that must be tackled by generating knowledge, for instance, through program comprehension. However, there are various causes for and forms of knowledge gaps. For instance, there may be an unawareness about knowledge that already exists, known knowledge gaps, unknowns that are not yet identified, or knowledge that is hidden from some people. Against this background, Israilidis et al. [13] argue that knowledge management should be seen as *ignorance management*. However, ignorance management as part of knowledge management requires a more detailed understanding of ignorance phenomena beyond non-knowledge being solely a lack of knowledge. Research in the (social-scientific) field of ignorance studies understands unknowns and knowledge gaps as social phenomena and considers the absence of knowledge not exclusively as a problem. *Ignorance* as a non-pejorative umbrella term is employed to grasp many related phenomena, such as secrets, strategic use of ignorance, intentional maintenance of knowledge gaps in decision making, or asymmetrical knowledge resulting from organizational structures.

In this paper, we argue that understanding ignorance as a social phenomenon can provide an alternative lens for software-engineering research to study knowledge gaps. To move in this direction, we first provide an overview of research in the field of ignorance studies that is relevant for studying software engineering. Based on this review, we suggest a working definition of



FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728503

non-knowledge. We employ the term non-knowledge (instead of ignorance) to denote phenomena related to knowledge gaps to make the relation to knowledge more visible. Then, we exemplify phenomena of non-knowledge in software and its engineering, discussing a vision for novel research through the lens of non-knowledge. We see the concept of non-knowledge as a highly valuable complement to account for the socio-technical and knowledge-intensive nature of software engineering—taking a perspective that has often been neglected in the past.

2 The Study of Non-Knowledge

Non-knowledge is the flip side of knowledge; both are inherently linked to each other [10]. Specifically, in the 1980s, Ravetz [28] asserted that growing knowledge is accompanied by an even swifter rise in what is not known. At this point, the discourse shifted towards advocating the management of non-knowledge, rather than assuming its continuous elimination through knowledge gain. In the following, we summarize related findings from ignorance studies. Our goal is not to be exhaustive, but rather to discuss the concepts we consider most relevant in the context of software engineering. We identified, sampled, and evaluated these concepts based on our backgrounds in social sciences and software engineering.

2.1 Non-Knowledge as a Social Construct

The study of non-knowledge in social sciences is not new. For example, Simmel [35] studied secrets in 1908, Moore and Tumin [24] researched social functions of ignorance in 1949, and Smithson [36] proposed a social theory of ignorance in 1985, just to name a few. Most importantly, such research has revealed that non-knowledge is socially constructed. This means that **non-knowledge does not just appear as an "accidental" byproduct of science, but is generated in social interaction** [42].

Over time, researchers studied social processes of producing or maintaining non-knowledge, as well as its negotiation and contestation by different groups of actors [2, 11]. In social processes, knowledge is sometimes intentionally hidden, and knowledge gaps willingly maintained to serve specific purposes (e.g., protecting intellectual property). Thus, **actors make strategic and intentional use of what is not known** [12, 22].

Besides such intentional uses, knowledge gaps can also be created unintentionally. For instance, Fleck [7] argued that non-knowledge is produced as a byproduct of knowledge generation due to specific epistemic practices, similar to what recently has been termed categorical blindness [17]. Also, knowledge asymmetries are related to specific forms of non-knowledge, such as tacit knowledge. Tacit knowledge refers to knowledge that a person possesses that cannot be easily communicated to others [3]. Essentially, social-sciences research has confirmed that **there are various causes that lead to unintentional non-knowledge**.

2.2 Dynamics of Non-Knowledge

Knowledge and non-knowledge are in a dynamic relationship. For instance, when answering open questions that are considered relevant, they are replaced by knowledge. In turn, new questions arise from that knowledge, producing non-knowledge. Simultaneously, knowledge can be forgotten by individuals or organizations, which results in new non-knowledge, too. Gross [10] has proposed a taxonomy to further grasp such dynamics by distinguishing between active and passive non-knowledge. Active non-knowledge is considered relevant by actors for their actions and decision making. This entails that actors deliberate on the limits of knowledge and can actively deal with these limits to answer open questions (e.g., defining research questions). In contrast, the term passive non-knowledge captures non-knowledge that is considered irrelevant by actors. The actors do not deliberate and even disregard the limits of knowledge in planning or decision making. Thus, although unknowns may be identified, they are considered irrelevant and do not generate further interest or attention and are not transformed into knowledge. In essence, **knowledge can be transformed into non-knowledge, and vice versa; depending on the interest of the actors involved in resolving the non-knowledge (i.e., active versus passive)**.

2.3 Categorizations of Non-Knowledge

Researchers have proposed many taxonomies to describe non-knowledge phenomena, presenting varying and partially conflicting classifications that are sometimes narrowed to the specific object being researched. As one taxonomy relating to the concepts we discussed (but which is by no means a one-to-one mapping), Wehling [42] suggests classifying non-knowledge along three dimensions:

Knowledge about Non-Knowledge (**KNK**, Section 2.1: social construction) represents a spectrum ranging from known unknowns (acknowledging that something is not known) to unknown unknowns (lack of awareness about not knowing), involving gradients like vaguely suspected or only partially conscious (non-)knowledge.

Intentionality of Non-Knowledge (**INK**, Section 2.2: active versus passive) refers to the intention to know or not to know something and to whether it is possible to attribute non-knowledge to (social) actors. It encompasses potentially deliberate avoidance of knowledge, as well as "weaker" manifestations like a lack of interest in acquiring knowledge or efforts to pursue knowledge that are quickly abandoned.

Assumed Temporal Permanence of Non-Knowledge (**ATPNK**, Section 2.2: transformation) covers possibilities and timing of transforming non-knowledge into knowledge. This property features a spectrum between the temporary state of "not-yet-knowing" to the seemingly insurmountable condition of "never-knowing-ability." Gradients include prolonged, but not fundamentally unsolvable states of not-knowing.

Using these dimensions instead of fixed categories, Wehling aims to emphasize the diversity and complexity of non-knowledge, while still covering the properties we discussed so far. Note that we use the abbreviations in parentheses to refer to these dimensions of non-knowledge in Section 3.2.

2.4 A Working Definition

As we exemplified in this section, non-knowledge has been extensively researched in social sciences. It encompasses various interconnected dimensions that have been redefined over time. Reflecting on this overview and the aspects we introduced, we propose the following working definition of non-knowledge:

Definition: Non-Knowledge

Non-knowledge refers to the absence of knowledge, which can be produced unintentionally or maintained intentionally. It encompasses the spectrum between recognized knowledge gaps and enduring unawareness, and is influenced by social factors that impact its potential transformation.

3 (Non-)Knowledge and Software

Modern software involves knowledge of various types about the software's design, development, and evolution (e.g., processes, people, code, tooling, architecture, dependencies, configuring). Managing such complex knowledge is challenging for many reasons. For example, even if developers can document knowledge to make it explicit, it is typically scattered across various artifacts and tools (e.g., models, code, comments, requirements, issue tracker, mailing lists). Additionally, such documentation is often not maintained and some knowledge may be restricted to certain people. Thus, knowledge is distributed asymmetrically (cf. Section 2.1). For such reasons, much software and software-engineering knowledge typically remains tacit and is forgotten, potentially contradicting other (documented) knowledge at some point, for instance, regarding changed processes, assumptions, intentions, or opinions. The following two examples are intended to illustrate the relevance of different forms of non-knowledge in software engineering.

3.1 Example: Linux Kernel

As a high-level example, the Linux Kernel can showcase the relation between complexity and non-knowledge. It is one of the most successful and largest software systems, which also builds the foundation for over 300 Linux distributions used in servers, smartphones, or embedded systems. The Kernel allows its users to configure (i.e., enable or disable) more than 14,000 unique functionalities, involves roughly 20 million lines of code, and exists for more than 30 years. Over this period, contributions to the Kernel have been made by around 14,000 developers and 1,200 organizations. It is evident that no developer can have full knowledge about more than the specific piece(s) of such a software that they work on.

3.2 Example: Marlin 3D Printer

As a more concrete example, we inspected pull request 9379[1] from the Marlin 3D-printer firmware to identify forms of non-knowledge. Marlin shares many commonalities with the Linux Kernel (e.g., same programming language, large community, open source). The pull request involves 162 comments, and our manual analysis revealed various types of non-knowledge and a diversity of strategies that developers relied on to deal with it. To showcase a few examples, we display an anonymized excerpt with respective numbers in Figure 1 (we refer to the categories by Wehling [42] that we introduced in Section 2.3 using bold labels):

① **Irrelevant Non-Knowledge:** In this comment, we can see that the developers are aware of their non-knowledge (i.e., are further tests needed?; **KNK**), but are unsure whether it is relevant for them or not (i.e., if the precision is good enough; **INK**). They may decide that there is no need to deal with existing unknowns (i.e., maybe the precision is good enough). This type of non-knowledge often becomes visible when developers gather potentially relevant knowledge, but it remains unclear to them what information is actually relevant or how it relates to their problem.

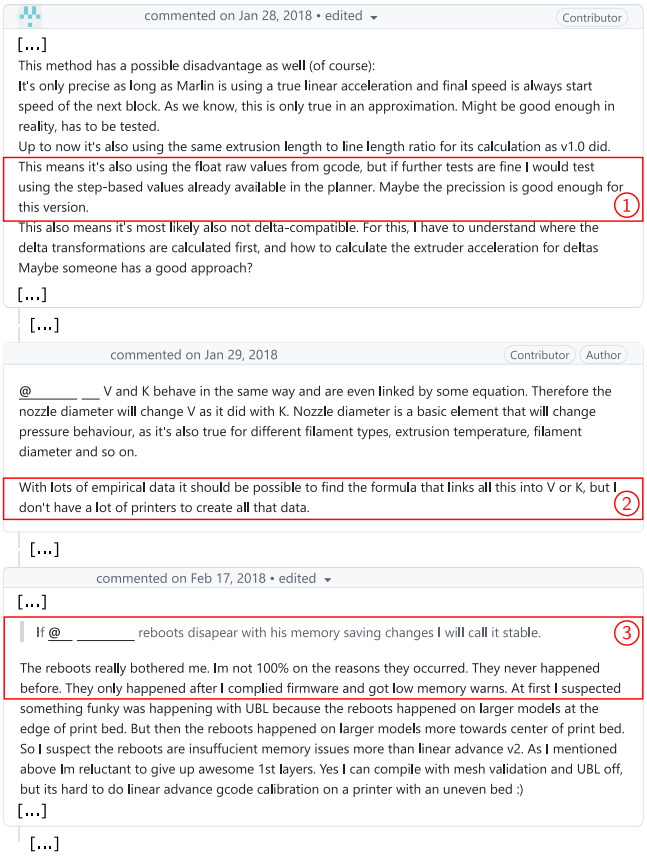


Figure 1: Anonymized examples of non-knowledge in Marlin pull request 9379.[1]

② **Known Non-Knowledge:** In this example, the developer explains why knowledge (data) cannot be created at that moment (**ATPNK**). The developer indicates that their expectations were not met and instead something different happened. Thus, they are aware of their non-knowledge as well as the means needed to find answers (i.e., empirical data to derive a formula; **KNK**). The preferred strategy to deal with (this type of) non-knowledge is to acquire knowledge. However, in this case, it became clear that technical equipment was required to do so; otherwise, the knowledge gaps would probably remain (**INK**). This example also hints at the relevance organizational structures (i.e., availability of tools for developers) can have for knowledge–non-knowledge dynamics.

③ **Unknown Non-Knowledge:** The category of unknown non-knowledge can only be identified in retrospect. In the example at hand, the developer reporting that they were surprised by the behavior of the software ("not 100 %") indicates that they were not aware about this non-knowledge beforehand

[1] https://github.com/MarlinFirmware/Marlin/pull/9379

(**KNK**). Specifically, the developer did not know what specific knowledge they lacked to identify the reasons for the reboots. Surprises often allow to pose new questions for specifying non-knowledge and subsequently generating the respective knowledge (**ATPNK**).

These are only first glimpses on types of non-knowledge that occur in software engineering. We argue that **we need systematic in-depth analyses to identify and understand types of non-knowledge** (e.g., related to code, documentation, decisions, requirements; relevant to what stakeholders; critical or confidential knowledge) to expand on temporary knowledge management; thereby moving towards ignorance management.

3.3 Temporary Knowledge Management

So far, research in software engineering typically understands non-knowledge as a problem that has to be tackled by generating knowledge. This understanding stems from research on program comprehension, which is developers' most frequent and expensive activity [39, 41]. Program comprehension is a complex activity in which a developer aims to understand a system's behavior and structure, primarily by reading the source code, but also by considering, for example, documentation, models, comments, or analysis tools [34]. This way, developers aim to tackle their non-knowledge by acquiring knowledge. Researchers in software engineering try to deal with non-knowledge by applying (semi-)automated techniques that help reverse engineer a specific type of information from certain artifacts [40]. For instance, different techniques exist to identify experts for a piece of code [23], locate features in source code [31], or provide on-demand documentation as soon as developers require knowledge [30] In general, **research in software engineering so far understands non-knowledge primarily as something that can and should be tackled by making knowledge available**.

4 Non-Knowledge as a New Lens

We propose to complement the current understanding of knowledge management in software engineering through the lens of non-knowledge. As we exemplified, non-knowledge in software engineering is diverse, including developers lacking knowledge about code (e.g., during on-boarding), forgetting knowledge (e.g., when a piece of code has not been maintained or when developers leave a project), having no reliable documentation (e.g., outdated), having (intentionally) no access to knowledge (e.g., information hiding), or missing tacit knowledge that is difficult to communicate [18–21, 26, 29]. Being able to understand, acknowledge the boundaries, and manage non-knowledge can expand developers' abilities to deal with software projects. For this purpose, we sketch three general directions for future software-engineering research.

Obtain a better understanding of the phenomenon of non-knowledge and its relevance (e.g., sustaining or transforming it). It is important to understand what types of non-knowledge exist in software engineering to identify causes of knowledge gaps and to address them in appropriate ways. Such research will allow, for example, to derive a more nuanced understanding of the phenomenon of tacit knowledge in the context of software engineering. Furthermore, for managing software projects, it will enable us to clearly specify knowledge gaps that are considered, for instance, relevant, irrelevant, intentional, or unintentional.

Derive nuanced insights into how knowledge is produced or is prevented from being produced. On the one hand, it is important to understand what strategies and means developers use to control the flow of knowledge, and what structures prevent the creation of knowledge. Since the relevance of non-knowledge is socially negotiated, it is important to understand to what extent and for what reasons different individuals (developers) consider non-knowledge as relevant or irrelevant; what strategies they use to deal with non-knowledge; as well as how their understanding and applied strategies are grounded in their social identities, values, or epistemic perspectives (e.g., developers employed by companies versus developers who contribute in their free time). On the other hand, it is relevant to understand factors that intervene and shape these processes; for example, group dynamics or organizational and cultural structures of a project (e.g., how groups share non-knowledge, hierarchies in projects).

Develop strategies for dealing with non-knowledge. Using the previous insights will allow us to improve the management of knowledge and non-knowledge in software projects. For instance, tackling knowledge gaps may reveal (non-)knowledge that was intended to be kept secret. Through a better understanding of knowledge–non-knowledge dynamics, appropriate strategies for dealing with knowledge gaps and managing non-knowledge within a project can be developed.

5 Conclusion

In this vision paper, we provided an overview of non-knowledge as a social phenomenon. As we have exemplified, non-knowledge is a recurring and diverse issue in software engineering. We are convinced that understanding and raising the awareness for non-knowledge can benefit software-engineering research and practice as well. To the best of our knowledge, current research does not reflect on the fact that non-knowledge is inherent to any complex software project, and may even be intended. Aiming to contribute to a new lens on software engineering, we have proposed three directions for future research.

By tackling these directions, we aim to advance software-engineering research and practice by working on a foundational understanding of non-knowledge and its related concepts. In essence, our take-away message for this vision paper is:

> **Non-Knowledge in Software Engineering**
>
> Developers constantly face non-knowledge about many possible aspects of their work, sometimes intentionally produced and sometimes not. We are convinced that in addition to understanding what we know (i.e., knowledge management), it is also important in software-engineering practice and research to reflect on what we do not know and for what reasons (i.e., non-knowledge) to enable ignorance management.

Acknowledgments

This research is supported by the German Research Foundation through project INKleSS (536290508).

References

[1] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM Transactions on Software Engineering and Methodology* 30, 4 (2021).

[2] Stefan Böschen and Peter Wehling. 2010. Introduction: Ambiguous Progress – Advisory and Regulatory Science between Uncertainty, Normative Disagreement and Policy-Making. *Science, Technology & Innovation Studies* 6, 2 (2010).

[3] Harry Collins. 2019. *Tacit and Explicit Knowledge.* University of Chicago.

[4] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Conference on Computer Supported Cooperative Work (CSCW).* ACM.

[5] Wei Ding, Peng Liang, Antony Tang, and Hans Van Vliet. 2014. Knowledge-Based Approaches in Software Documentation: A Systematic Literature Review. *Information and Software Technology* 56, 6 (2014).

[6] Christof Ebert. 2008. Open Source Software in Industry. *IEEE Software* 25, 3 (2008).

[7] Ludwik Fleck. 1981. *Genesis and Development of a Scientific Fact.* University of Chicago Press.

[8] Beat Fluri, Michael Würsch, and Harald C. Gall. 2007. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *Working Conference on Reverse Engineering (WCRE).* IEEE.

[9] Tom Groot, Lina Ochoa Venegas, Bogdan Lazăr, and Jacob Krüger. 2024. A Catalog of Unintended Software Dependencies in Multi-Lingual Systems at ASML. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* ACM.

[10] Matthias Gross. 2019. The Paradox of the Unexpected: Normal Surprises and Living with Nonknowledge. *Environment: Science and Policy for Sustainable Development* 61, 3 (2019).

[11] Stephen Hilgartner. 2001. Election 2000 and the Production of the Unknowable. *Social Studies of Science* 31, 3 (2001).

[12] Stephen Hilgartner. 2012. Selective Flows of Knowledge in Technoscientific Interaction: Information Control in Genome Research. *The British Journal for the History of Science* 45, 2 (2012).

[13] John Israilidis, Lock Russell, and Louise Cooke. 2013. Ignorance Management. *Management Dynamics in the Knowledge Economy* 1, 1 (2013).

[14] Eirini Kalliamvakou, Christian Bird, Thomas Zimmermann, Andrew Begel, Robert DeLine, and Daniel M. German. 2019. What Makes a Great Manager of Software Engineers? *IEEE Transactions on Software Engineering* 45, 1 (2019).

[15] Rashidah Kasauli, Eric Knauss, Jennifer Horkoff, Grischa Liebel, and Francisco G. de Oliveira Neto. 2021. Requirements Engineering Challenges and Practices in Large-Scale Agile System Development. *Journal of Systems and Software* 172 (2021).

[16] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *International Coference on Mining Software Repositories (MSR).* IEEE.

[17] Morten Knudsen. 2011. Forms of Inattentiveness: The Production of Blindness in the Development of a Technology for the Observation of Quality in Health Services. *Organization Studies* 32, 7 (2011).

[18] Jacob Krüger and Regina Hebig. 2020. What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In *International Conference on Software Maintenance and Evolution (ICSME).* IEEE.

[19] Jacob Krüger and Regina Hebig. 2023. To Memorize or to Document: A Survey of Developers' Views on Knowledge Availability. In *International Conference on Product Focused Software Process Improvement (PROFES).* Springer.

[20] Jacob Krüger, Sebastian Nielebock, and Robert Heumüller. 2020. How Can I Contribute? A Qualitative Analysis of Community Websites of 25 Unix-Like Distributions. In *International Conference on Evaluation and Assessment in Software Engineering (EASE).* ACM.

[21] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do You Remember This Source Code?. In *International Conference on Software Engineering (ICSE).* ACM.

[22] Linsey McGoey. 2007. On the Will to Ignorance in Bureaucracy. *Economy and Society* 36, 2 (2007).

[23] Audris Mockus and James D. Herbsleb. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *International Conference on Software Engineering (ICSE).* ACM.

[24] Wilbert E. Moore and Melvin M. Tumin. 1949. Some Social Functions of Ignorance. *American Sociological Review* 14, 6 (1949).

[25] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting Source Code: Is It Worth It for Small Programming Tasks? *Empirical Software Engineering* 24, 3 (2019).

[26] Chris Parnin. 2010. A Cognitive Neuroscience Perspective on Memory for Programming Tasks. In *Annual Workshop of the Psychology of Programming Interest Group (WPPIG).* PPIG.

[27] Chris Parnin and Spencer Rugaber. 2012. Programmer Information Needs after Memory Failure. In *International Conference on Program Comprehension (ICPC).* IEEE.

[28] Jerome R Ravetz. 1987. Usable Knowledge, Usable Ignorance: Incomplete Science with Policy Implications. *Knowledge* 9, 1 (1987).

[29] Andreas Riege. 2005. Three-Dozen Knowledge-Sharing Barriers Managers Must Consider. *Journal of Knowledge Management* 9, 3 (2005).

[30] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil A. Ernst, Marco A. Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. 2017. On-Demand Developer Documentation. In *International Conference on Software Maintenance and Evolution (ICSME).* IEEE.

[31] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*, Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin (Eds.). Springer.

[32] Ioana Rus, Mikael Lindvall, and S Sinha. 2002. Knowledge Management in Software Engineering. *IEEE Software* 19, 3 (2002).

[33] Kurt Schneider. 2009. *Experience and Knowledge Management in Software Engineering.* Springer.

[34] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending Studies on Program Comprehension. In *International Conference on Program Comprehension (ICPC).* IEEE.

[35] Georg Simmel. 1908. *Soziologie: Untersuchungen über die Formen der Vergesellschaftung.* Duncker & Humblot. In German.

[36] Michael Smithson. 1985. Toward a Social Theory of Ignorance. *Journal for the Theory of Social Behaviour* 15, 2 (1985).

[37] Vanessa Sochat. 2021. The 10 Best Practices for Remote Software Engineering. *Communications of the ACM* 64, 5 (2021).

[38] Margaret-Anne Storey, Neil A. Ernst, Courtney Williams, and Eirini Kalliamvakou. 2020. The Who, What, How of Software Engineering Research: A Socio-Technical Framework. *Empirical Software Engineering* 25, 5 (2020).

[39] Rebecca Tiarks. 2011. What Maintenance Programmers Really Do: An Observational Study. In *Workshop on Software Reengineering (WSR).* University of Siegen.

[40] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. 2007. Empirical Studies in Reverse Engineering: State of the Art and Future Trends. *Empirical Software Engineering* 12, 5 (2007).

[41] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer* 28, 8 (1995).

[42] Peter Wehling. 2024. Nichtwissen – ein ungewöhnlicher Schlüsselbegriff der Umweltsoziologie. In *Handbuch Umweltsoziologie*, Marco Sonnberger, Alena Bleicher, and Matthias Groß (Eds.). Springer. In German.