# Tackling Knowledge Needs during Software Evolution

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany
jkrueger@ovgu.de

## ABSTRACT

Developers use a large amount of their time to understand the system they work on, an activity referred to as program comprehension. Especially software evolution and forgetting over time lead to developers becoming unfamiliar with a system. To support them during program comprehension, we can employ knowledge recovery to reverse engineer implicit information from the system and the platform (e.g., GitHub) it is hosted on. However, to recover useful knowledge and to provide it in a useful way, we first need to understand what knowledge developers forget to what extent, what sources are reliable to recover knowledge, and how to trace knowledge to the features in a system. We tackle these three issues, aiming to provide empirical insights and tooling to support developers during software evolution and maintenance. The results help practitioners, as we support the analysis and understanding of systems, as well as researchers, showing opportunities to automate, for example, reverse-engineering techniques.

## CCS CONCEPTS

• **Software and its engineering** → *Software design tradeoffs*; *Maintaining software*; • **Applied computing** → *Psychology*.

## KEYWORDS

Program comprehension, Feature traceability, Forgetting, Memory, Software maintenance, Software evolution

**ACM Reference Format:**
Jacob Krüger. 2019. Tackling Knowledge Needs during Software Evolution. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3338906.3342505

## 1 INTRODUCTION

Software developers spend most of their time on program comprehension [50] to understand a system's features and evolution [8, 38, 52–54]. During program comprehension, developers aim to improve or recover their knowledge about the features and their characteristics (so-called facets [7]) that are relevant for their task; in particular, locating the code that belongs to a feature [11, 26, 44, 54]. To support knowledge recovery, several techniques have been proposed to either facilitate the understanding of code (e.g. clean code guidelines [33]), to extract knowledge (e.g. feature location [11, 44]) or to store knowledge about a system (e.g. code comments [39]).

In parallel, *software-hosting platforms*, such as GitHub [20], are used more and more often for software development. Such platforms integrate a version control system with other capabilities to track a system's evolution and corresponding knowledge, for example, Wiki and discussion pages—which is why we partly focus on these platforms. Considering the evolution of software on such platforms, we focus on three research questions:

**RQ$_1$** What knowledge do developers remember?
**RQ$_2$** What sources exist to recover knowledge?
**RQ$_3$** How can we trace knowledge to features?

For **RQ$_1$**, we adopt psychological research on forgetting to understand what knowledge (e.g., code, architecture, features) developers remember at what point in time. Concerning **RQ$_2$**, we identify suitable information sources in software-hosting platforms to recover knowledge that may be forgotten. During **RQ$_3$**, we investigate techniques to trace knowledge directly to the source code to improve its accessibility for the developer.

## 2 STATE OF THE ART

Extensive research focuses on techniques and artifacts to understand and support program comprehension [46], for example, cognitive models of the understanding of code [50, 53], analysis of neural aspects [17, 47], and improvements of source-code readability [1, 15]. Moreover, several techniques have been proposed to recover implicit knowledge from systems based on reverse engineering [9, 37] and to better trace the features in a system to enrich the available knowledge [13, 18]. While these aspects have been studied extensively, software evolution adds different perspectives that have rarely been analyzed in detail. For example, it is important to understand what developers may still remember about a system and its features, and what knowledge they actually need [3, 19, 40, 48]. However, little research focuses on understanding how developers forget what type of knowledge and how we can support them to recover this knowledge [21]. There has been extensive research on program comprehension and knowledge recovery (e.g., expertise identification [14, 34]), but the current practices need to be improved [43]. In particular, we need to enable and benchmark automated techniques [51] that provide reliable knowledge to developers according to their needs—which is further highlighted by several visions, for example, of Berger [6], Murphy [35] or Robillard et al. [42]. For this purpose, we aim to improve our understanding of how developers forget and we plan to provide capabilities to recover and trace knowledge to features in a systematic way. To achieve our goals, we build on and extend existing techniques that aim to recover and present knowledge to software engineers, facilitating their program comprehension, and thus tasks.

## 3 RESEARCH APPROACH

Our work focuses on empirical research, aiming to understand how developers forget what knowledge. Particularly in this regard, we are cross-cutting with the domain of psychology and adopt their concepts on forgetting curves and memory for software engineering. Moreover, we analyze systems and processes, open-source as well as industrial, to understand the needs for and the availability of knowledge, for example, in software-hosting platforms. In order to tackle our goals, we adopt various established and new research methods. For instance, we conduct interview studies and systematic literature reviews to collect existing evidence and perform case studies as well as controlled experiments to evaluate new techniques [23, 55]. To gain a broader practical perspective, we conduct empirical studies with developers in community-question answering systems [49], such as Stack Overflow, as these provide a large, collective knowledge base [31].

## 4 CONTRIBUTIONS

In this section, we briefly summarize our current findings for each research question and then sketch our ongoing and future work.

**RQ$_1$: Measuring Knowledge.** Research on program comprehension focuses heavily on improving the actual source code of a system [46]. This seems reasonable, as most developers tend to believe the source code more than external information that are provided, for example, as comments [39, 43]. However, the question arises, what knowledge developers still have about their source code after what time. To this end, we conducted an empirical study [32] with 60 open-source developers and identified factors that influence their memory, derived their memory strength, and analyzed the established forgetting curve of Ebbinghaus [12]. We could show that especially repetitions and own code positively impact memory performance. Without considering these factors, developers forget the source code exponentially, as defined in the forgetting curve, which we aim to adapt to consider these factors.

**RQ$_2$: Recovering Knowledge.** As developers forget or may be new to a system, it is necessary to understand what sources we can exploit to recover knowledge, for which we focus on software-hosting platforms. Despite such platforms, especially feature location is still a largely manual process [26]. Even for cloned or forked system that developed apart from a common base, automation is hardly achievable, although we can identify code clones to analyze their differences and commonalities. However, these do not align to actual features [30]. The problem is that features are rarely mapped explicitly to the code and their additional facets, for example, how they got approved or were tested, are usually implicit.

We analyzed two systems from GitHub to identify what facets that are relevant to evolve features of a system [7] we can recover from what sources [29]. During this analysis, we found that modern software-hosting platforms provide several information sources that we can exploit for various facets to some extent. For example, the release log of one system helped us to identify some features and was mapping them to the source code, due to linking to pull requests and commits. However, this knowledge is implicit and needs to be stored and made explicit by reverse engineering it.

**RQ$_3$: Tracing Knowledge.** Finally, we are concerned with tracing knowledge back to features and the corresponding source code, allowing developers to access it on demand. Still, as also our results on trust in information show [39], we need sources that developers do trust for this purpose. Software-hosting platforms are a helpful means in this regard, as they automatically track changes and store additional artifacts, such as discussions. In our current research [24], we are aiming to establish a tracing between the code and features by adding annotations in the source code that are automatically synchronized with a feature model [5, 10, 45].

To support our technique, we conducted multiple empirical studies, collecting principles on how developers locate features [26] and construct a feature model [36]. We plan to combine and automate our insights to maintain the feature model and its mapping to the code. For this mapping, we decided to use annotations in the code, as we found that they can benefit program comprehension [27]. Practitioners also argue, and our results confirm, that decomposing features into separate classes can be more challenging than annotations (depending on the granularity [22]), as the system's behavior is split up and not understandable anymore, referred to as action at a distance [25, 28]. Moreover, support to automatically maintain, trace, and verify annotations has already been proposed and developed, allowing us to build on existing research [2, 4].

**Ongoing Work.** Currently, we develop a concept to refine and combine our previous results. First, we are conducting additional studies to verify our findings. In particular, we focus on understanding and categorizing what knowledge developers forget and remember. For this purpose, we conduct interview studies and experiments to analyze whether developers recall other abstractions of their system (e.g., architectures, features) longer then the source code. This can help us to gain insights into what developers consider important and how they remember, as well as what knowledge we need to recover and present to them in more detail.

Second, we implement tools that automatically recover knowledge from systems and their platforms, combine it, and present it to the developer. We intend to show how we can use feature traces (i.e., annotations) to more accurately provide relevant knowledge on the system's evolution. To this end, we plan to conduct experiments, simulations, and case studies especially on preprocessor-based systems, such as the Linux Kernel. Preprocessors add annotations that are used to configure systems [5] and partly align to features. Also, the evolutionary history (e.g. of Linux [16, 41]) provides a useful data source to evaluate our techniques further. Overall, our goal is to provide a technique that automatically recovers knowledge for a system, traces it to the system's features, and presents it to the developer who is working on that feature on demand.

## 5 CONCLUSION

Changes in a system challenge developers' program comprehension and require them the recover implicit knowledge. In this paper, we reported results we obtained from various empirical studies on program comprehension during software evolution. We highlighted that knowledge needs can change during software evolution, for example, due to forgetting and unnoticed changes, Moreover, we described how we intend to extend our current findings and provide automation that supports developers during their tasks. This way, we hope to facilitate software development and maintenance, tackling the fact that developers forget and software evolves.

## REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An Empirical Study of the Impact of two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *CSMR*.

[2] Hadil Abukwaik, Andreas Burger, Berima K. Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *ICSME*.

[3] Abdullah Al-Nayeem, Krzysztof Ostrowski, Sebastian Pueblas, Christophe Restif, and Sai Zhang. 2017. Information Needs for Validating Evolving Software Systems: An Exploratory Study at Google. In *ICST*.

[4] Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. 2017. FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces. In *VaMoS*.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[6] Thorsten Berger. 2017. Feature-Oriented Traceability. In *Grand Challenges of Traceability: The Next Ten Years*.

[7] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature?: A Qualitative Study of Features in Industrial Software Product Lines. In *SPLC*.

[8] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program Understanding and the Concept Assignment Problem. *Communications of the ACM* (1994).

[9] Elliot J. Chikofsky and James H. Cross. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* (1990).

[10] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*.

[11] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process* (2013).

[12] Hermann Ebbinghaus. 1885. *Über das Gedächtnis: Untersuchungen zur Experimentellen Psychologie.* Duncker & Humblot. In German.

[13] Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, and Christophe Dony. 2013. Feature-to-Code Traceability in a Collection of Software Variants: Combining Formal Concept Analysis and Information Retrieval. In *IRI*.

[14] Thomas Fritz, Gail C. Murphy, and Emily Hill. 2007. Does a Programmer's Activity Indicate Knowledge of Code?. In *ESEC/FSE*.

[15] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2019. Shorter Identifier Names take Longer to Comprehend. *Empirical Software Engineering* (2019).

[16] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux Kernel as a Case Study in Software Evolution. *Journal of Systems and Software* (2010).

[17] Ahmad Jbara and Dror G. Feitelson. 2017. How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking. *Empirical Software Engineering* (2017).

[18] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*.

[19] Jitendra Josyula, Sarat Panamgipalli, Muhammad Usman, Ricardo Britto, and Nauman Bin Ali. 2018. Software Practitioners' Information Needs and Sources: A Survey Study. In *IWESEP*.

[20] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *MSR*.

[21] Keumseok Kang and Jungpil Hahn. 2009. Learning and Forgetting Curves in Software Development: Does Type of Knowledge Matter? *ICIS Proceedings* (2009).

[22] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *ICSE*.

[23] Barbara A. Kitchenham, David Budgen, and Pearl Brereton. 2016. *Evidence-Based Software Engineering and Systematic Reviews.* CRC Press.

[24] Sebastian Krieter, Jacob Krüger, and Thomas Leich. 2018. Don't Worry About It: Managing Variability On-the-Fly. In *VaMoS*.

[25] Jacob Krüger. 2018. Separation of Concerns: Experiences of the Crowd. In *SAC*.

[26] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. Features and How to Find Them: A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems*.

[27] Jacob Krüger, Gül Calıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *ESEC/FSE*.

[28] Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. 2018. Physical Separation of Features: A Survey with CPP Developers. In *SAC*.

[29] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* (2019).

[30] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *SPLC*.

[31] Jacob Krüger, Ivonne Schröter, Andy Kenner, and Thomas Leich. 2017. Empirical Studies in Question-Answering Systems: A Discussion. In *CESI*.

[32] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do You Remember This Source Code?. In *ICSE*.

[33] Robert C. Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson.

[34] Audris Mockus and James D. Herbsleb. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *ICSE*.

[35] Gail C. Murphy. 2019. Beyond Integrated Development Environments: Adding Context to Software Development. In *ICSE*.

[36] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *ESEC/FSE*.

[37] Joaquín Nicolás and Ambrosio Toval. 2009. On the Generation of Requirements Specifications from Software Engineering Models: A Systematic Literature Review. *Information and Software Technology* (2009).

[38] Michael Nieke, Lukas Linsbauer, Jacob Krüger, and Thomas Leich. 2019. Second Intl. Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution). In *SPLC*.

[39] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2018. Commenting Source Code: Is It Worth It For Small Programming Tasks? *Empirical Software Engineering* (2018).

[40] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction* (2018).

[41] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco T. Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *MODULARITY*.

[42] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David C. Shepherd, and Edmund Wong. 2017. On-Demand Developer Documentation. In *ICSME*.

[43] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How Do Professional Developers Comprehend Software?. In *ICSE*.

[44] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*.

[45] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *RE*.

[46] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending Studies on Program Comprehension. In *ICPC*.

[47] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes C. Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring Neural Efficiency of Program Comprehension. In *ESEC/FSE*.

[48] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask During Software Evolution Tasks. In *FSE*.

[49] Ivan Srba and Maria Bielikova. 2016. A Comprehensive Survey and Classification of Approaches for Community Question Answering. *ACM Transactions on the Web* (2016).

[50] Margaret-Anne Storey. 2006. Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future. *Software Quality Journal* (2006).

[51] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *SPLC*.

[52] Rebecca Tiarks. 2011. What Maintenance Programmers Really do: An Observational Study. In *WSR*.

[53] Anneliese Von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer* (1995).

[54] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: a Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process* (2013).

[55] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering.* Springer.