Do You Remember This Source Code?

Jacob Krüger
Harz University of Applied Sciences
Otto-von-Guericke-University
Wernigerode & Magdeburg, Germany
jkrueger@ovgu.de

Jens Wiemann
Otto-von-Guericke-University
Magdeburg, Germany

Wolfram Fenske
Otto-von-Guericke-University
Magdeburg, Germany
wfenske@ovgu.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
saake@ovgu.de

Thomas Leich
Harz University of Applied Sciences
METOP GmbH
Wernigerode & Magdeburg, Germany
tleich@hs-harz.de

ABSTRACT

Being familiar with the source code of a program comprises knowledge about its purpose, structure, and details. Consequently, familiarity is an important factor in many contexts of software development, especially for maintenance and program comprehension. As a result, familiarity is considered to some extent in many different approaches, for example, to model costs or to identify experts. Still, all approaches we are aware of require a manual assessment of familiarity and empirical analyses of *forgetting* in software development are missing. In this paper, we address this issue with an empirical study that we conducted with 60 open-source developers. We used a survey to receive information on the developers' familiarity and analyze the responses based on data we extract from their used version control systems. The results show that forgetting is an important factor when considering familiarity and program comprehension of developers. We find that a forgetting curve is partly applicable for software development, investigate three factors – the number of edits, ratio of owned code, and tracking behavior – that can impact familiarity with code, and derive a general memory strength for our participants. Our findings can be used to scope approaches that have to consider familiarity and they provide insights into forgetting in the context of software development.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Maintaining software**; Risk management; • **Applied computing → Psychology**;

KEYWORDS

Familiarity, forgetting, empirical study, maintenance, program comprehension, expert identification, knowledge management

ACM Reference Format:
Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do You Remember This Source Code?. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180215

1 INTRODUCTION

Developers' familiarity (or expertise) with a project's context – comprising programs and colleagues – is an essential factor for many aspects of software engineering, such as, team and task performance [16, 29, 37], knowledge sharing [37, 54, 67], and tool acceptance [17, 37, 54]. Considering the software itself, familiarity influences how fast and reliable developers can comprehend, enhance, and maintain a program, for instance, to locate and fix bugs or for reengineering [7, 55, 60, 62]. Consequently, *software familiarity*, comprising the knowledge on a programs' source code, design, and usage, facilitates maintenance tasks [2, 66]. Especially as maintaining software is the main cost driver in software development [8, 12, 23, 61, 64], comprehending and familiarizing with a program is essential [60]. For this reason, familiarity is considered as an important factor in many cost estimation approaches [1, 9–11, 38] and identifying experts for a piece of code receives much attention [19, 46–48].

While team familiarity has been investigated extensively [28, 29, 43, 52], less research focuses on analyzing software familiarity. The main issue in this context are developers forgetting details about their source code, complicating software development and maintenance [33, 39, 66]. To address this issue, approaches on *program comprehension* aim to support developers in regaining their familiarity. Several approaches, such as, clean code guidelines [44] or suitable identifier names [27, 41, 65], have been analyzed and proposed to improve the comprehension of source code [57]. However, at this point familiarity must already be regained [4, 34]. Assessing how familiar a developer still is with the source code is essential, for example, to assign tasks, to identify experts, or, consequently, to reduce and estimate costs.

In this paper, we propose to adopt forgetting curves [4, 32, 50] from the psychological domain for software engineering. For this purpose, we utilize the forgetting curve proposed by Ebbinghaus [15] and test its applicability for software engineering. Thus, the main focus of our work is an empirical study that we conducted with 60 open-source developers from 10 GitHub projects. They participated in an online survey in which they, for example, approximated their own familiarity with a specific file. Based on their responses and data from their commits, we investigate three factors – namely, the number of repeated edits, ratio of own code, and tracking behavior – that may affect their familiarity, derive an average memory strength, and test the aforementioned forgetting curve. Our findings help to understand how developers forget details about their source code and how to estimate their remaining familiarity. The results can support many approaches in software engineering, for example, to allocate developers to tasks they are most efficient on, to identify experts, and to search for knowledge gaps in a project. Moreover, reliable familiarity estimations can improve the accuracy of cost models. In detail, we contribute the following in this paper:

- We report an empirical study that we conducted as an online survey. Based on 60 responses with open source developers, we investigate the importance of repetition, ratio of own code, and code tracking on their familiarity. The results indicate moderate to strong correlations for the first two factors. Surprisingly, we find no correlation between familiarity and tracking for our participants.
- We identify an average memory strength for our participants as a crucial factor to approximate forgetting. While this value needs to be refined in the future, it provides hints at how fast developers may forget their code and on the reliability of self-assessments. Also, researchers and practitioners can use this value as baseline for further research.
- We test if Ebbinghaus' [15] forgetting curve is applicable in software engineering. For this purpose, we analyze which of the investigated factors distort the standard course of the curve. The results show that different factors need to be considered before the forgetting curve can be fully applied. Nonetheless, if these factors are not effective, the curve actually fits well to the responses of our participants.

Overall, we aim at analyzing the effects of forgetting to derive approaches to automatically measure or improve software familiarity in future research.

2 BACKGROUND

In this section, we introduce background information on *familiarity* and *forgetting curves*. Both concepts are essential for the understanding of this paper.

2.1 Familiarity

Familiarity comprises knowledge persons gain on different aspects of their daily work and about their team members. Over time, they become familiar with their domain and each other, improving interactions, implementing a knowledge base, and supporting the identification of expertise [50, 52]. Thus, studies show positive effects of familiarity on, for example, team performance in flight simulations, problem solving, and several other tasks [16, 42, 43].

In software development, this familiarity focuses on a specific program and the developing team. While it is an important factor, we are not aware of detailed analyses and measurements of a developer's familiarity with a program. For example, Boehm et al. [10] introduce a scale to measure unfamiliarity for cost estimations with the COCOMO II model and its extensions [5, 11]. The proposed scale ranges from 0 to 1, representing *completely familiar* and *completely unfamiliar*, respectively. Still, the actual value must be judged by a user.

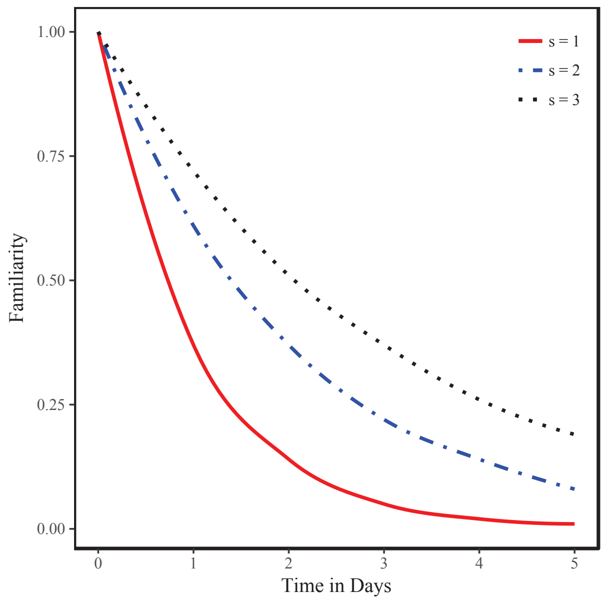

Figure 1: Forgetting curves of Ebbinghaus [15] for memory strengths (s) of 1, 2, and 3.

In the context of this work, we are focusing on a developer's familiarity with a program. This *software familiarity* is the result of studying and working with a program, leading to knowledge about the purpose, usage, and structure, for example of a file. We are aware that several terms exist that are closely related, used synonymously, and sometimes may be interpreted in the same way, for instance, comprehension, knowledge, learning, expertise, or experience. However, we rely on the term familiarity in this paper, as it subsumes such meanings.

2.2 Forgetting

Familiarity is no consistent state: It can be gained but also fades over time. The main reason for becoming unfamiliar with a program is that we *forget* details about it. Consequently, over time developers become less familiar with their code and need more effort to regain the necessary familiarity to work on it. This forgetting process basically represents the opposite and, thus, is strongly connected to learning [32, 50]. In psychology, different forgetting models and curves exist [32, 50]. We rely on the forgetting curve described by Ebbinghaus [15]. While it is rather old, studies show that it can be replicated and performs similar to other curves [4, 49]. In the context of this work, the remaining memory calculated with this curve represents the developers' familiarity.

Ebbinghaus [15] describes an exponential nature of forgetting, as we display in Equation 1.

$$R = e^{-\frac{t}{s}} \qquad (1)$$

Here, R represents the resulting memory retention rate. The relative memory strength of the considered subject is defined by s and t is the time (in days) between studying an artifact and testing the subject's memory. To exemplify this function, we display curves for three different memory strengths in Figure 1. As we can see, a higher memory strength results in a slower retention rate, meaning that the familiarity remains for a longer time. For example, with a memory strength of $s = 1$ (solid red line in Figure 1), after the first day only 37% of the familiarity remains. In contrast, a memory strength of $s = 3$ (dotted black line in Figure 1) indicates that the same value is reached only after three days. This memory strength is individual for each person and depends on several factors, such as, learning effects and the considered artifact.

3 PROBLEM STATEMENT

In several scientific and industrial domains it is essential to consider the familiarity of software developers with the code they work on. There are multiple factors that influence how fast developers forget and, thus, loose familiarity with a program. Consider the following example with two developers, A and B: At first, developer A implements a file and at the point of creation is most likely completely familiar with it. However, when he stops working on the file for some time, his familiarity decreases, potentially resembling a forgetting curve by Ebbinghaus [15], as we depict in Figure 1. Additionally, developer B changes the file, for example, to add new functions or remove bugs. Here, two other factors besides forgetting apply: Developer B has to understand the existing code at least far enough to change it and, thus, gains familiarity. He could even gain 100% familiarity if he would investigate every detail of the file. In contrast, the familiarity of developer A is negatively affected because he also has to analyze the new implementation at some point. Any further change results in the same effects and in one developer loosing familiarity while the other may gain it.

This example raises several questions regarding the impact of code changes on a developer‘s familiarity, for example: *Is a forgetting curve appropriate for software developers? How fast do software developers actually forget their source code? Do repeated commits improve the memory strength? How do changes of other developers impact familiarity? How many changes of others do developers analyze? Which other tasks, such as, reviewing or testing, affect familiarity?* In this work, we focus on a subset of these questions. Namely, we investigate the applicability of Ebbinghaus' [15] forgetting curve on software developers, their average memory strength, and if repetitions, the ratio of own code, or observing others' changes on own files affect familiarity. Despite our focus on these factors, all stated questions are important future work. The resulting findings can be used to derive approaches for measuring familiarity and to improve our understating of software engineering activities.

To this end, we derive the following three scenarios that affect familiarity based on our example:

Sc_1 *Forget:* Over time, developers lose knowledge and become less familiar with source code they worked on. Thus, they cannot recall all details anymore and need time to regain familiarity.

Sc_2 *Gain:* Developers who edit source code, for instance, by adding, modifying, or removing lines, aim to understand already existing code in addition to their newly written code. Thus, they gain or regain familiarity, due to analyzing existing source code, either someone else's or their own.

Sc_3 *Unaware:* If another developer edits source code, the original author is unlikely to review the modifications until it is necessary. For this reason, the familiarity of a developer decreases with any edit another one applies, due to his unawareness of the modification.

In the remaining paper, we refer to these scenarios to describe which aspects of familiarity we address with our analysis.

4 SURVEY DESIGN

To address the aforementioned questions, we conducted an online survey. In this section, we describe our *research questions*, *survey setup*, and *subjects*.

4.1 Research Questions

The goal of our survey is to provide insights into software familiarity and especially on factors that preserve it. Thus, we aim to answer the following research questions:

RQ_1 Do the factors repetition, ratio of own code, or change tracking affect a developer's familiarity?
There are many factors that can affect a developer's familiarity, of which we investigate three: Firstly, we hypothesize that repeatedly working on the same code refreshes familiarity (Sc_2) and improves capabilities to remember details, due to learning effects (Sc_1). Based on the results of Glenberg [21], we expect a monotonically rising dependency. Secondly, developers should also be more familiar with a file if they wrote a larger ratio of it (Sc_3). Finally, we analyze whether our participants track changes (Sc_3) of other developers on their files and if this affects their familiarity. Answering this research question will contribute empirical findings regarding the influence of these factors on software familiarity.

RQ_2 What is the average memory strength of a developer regarding the source code?
As we described in Section 2, the memory strength is necessary to estimate how fast forgetting proceeds (Sc_1). Thus, our results help to approximate how long developers remember code, supporting corresponding estimations. We remark that this factor is heavily impacted by individuals' characteristics and can also be trained. Consequently, there can be considerable outliers for a specific developer.

RQ_3 Is Ebinghaus' forgetting curve applicable for software developers?
In the end, we aim to assess the applicability of Ebbinghaus' [15] forgetting curve in software development. For this purpose, we compare the self-evaluations of our participants with computed values. The outcome helps to design further research, for example, to automatically measure familiarity or improve approaches that are based on it.

Overall, answering these questions helps research and industry alike to analyze, understand, and improve software development. Our findings support many research areas, such as, cost estimation, knowledge management, teaching programming, expert identification, and reengineering.

4.2 Survey Setup

For the general setup of our study, we decided to perform an online survey. As an introducing part of the survey we provided the following short introduction of familiarity to avoid confusion on this term:

> Software familiarity – generally known as a result of study or experience. If familiar, you know: The purpose of a file, its usage across the project, and its structure or programming patterns.

We asked to insert the GitHub username or mail address to prevent multiple responses from the same developer and to track their commits. Furthermore, all developers had to specify one file of their project they had been working with. Here, we especially asked them not to check this file before participating. We used several questions, of which the following are of interest for this work.

How well do you know the content of the file? For this question, the developers have to assess how familiar they are with the file they specified before. Here, they can rate their familiarity on a Likert scale from 1 (i.e., barely the purpose) to 9 (i.e., purpose, usage, and structure), which represents percentages. We do not allow a rating of 0 as the developer of a file should have at least some knowledge about it. Furthermore, we assume that participants do not know all details (e.g., each line) of the code even if they developed it and, thus, we do not allow a 10. We use the answers for all our research questions, as they provide the basis for our analysis.

After how many days do you only remember the structure and purpose of a file, but have forgotten the details? In this case, the participants have to estimate after how many days they would still have a remaining familiarity of 5 (i.e., its purpose and usage). While this is a challenging estimation, we use the answers to validate our calculations for the second research question.

How well do you track changes other developers make on your files? This question addresses our first research question. We can extract values for repetitions based on commits and the ratio of own code based on a file's history. In contrast, we have to personally ask our participants whether they track changes that others do on their files. To this end, we use a Likert scale ranging from 0 (i.e., no tracking at all) to 10 (i.e., analyzing each change).

How many lines of code does the file contain? We ask this question to validate whether the participants remember the correct file or are just too unfamiliar with it. Here, we exclude responses with a high error rate, as we describe in the next section.

When was the last date you edited the file? Again, we use this question to validate the participants' responses. A high deviation from the real date of the last edit may indicate missing motivation or a wrong file being remembered. Thus, we also exclude such responses from our analysis.

4.3 Subjects

As we aimed to use data from version control systems to answer our research questions, we considered the ten GitHub projects we display in Table 1. We varied our selection to consider different development approaches and to increase the response rate, wherefore we included projects with different attributes: Firstly, we searched for actively developed and popular projects from which we invited all developers that edited a file in 2016. Secondly, we varied the programming language and team size to consider different development styles. Finally, we considered some scientific projects, for example `astropy`, as research shows that response rates for these are higher [14]. Each active developer received a mail containing a link to our survey.

Table 1: Projects considered for the survey.

Project	Language	Developers		
		Inv.	Resp.	Incl.
aframe	JavaScript	43	5	4
angular.js	JavaScript	75	8	7
astropy	Python	41	13	7
ember.js	JavaScript	75	5	3
FeatureIDE	Java	10	4	4
ipython	Python	33	3	3
odoo	Python	135	15	10
react	JavaScript	153	4	4
serverless	JavaScript	89	12	11
sympy	Python	68	9	7
Overall		722	78	60

Inv:: Invited; Resp.: Responded; Incl.: Included

To consider the quality of responses, we define the following exclusion criteria:

(1) *Participant did not edit the selected file:* As we do not ask the participants to specify a file they committed to – but with which they worked – 4 of them picked one they did not commit to. As we cannot extract any information from the commits, we remove these responses.
(2) *Last edit was more than a year ago:* We especially ask the participants to specify a file they worked on in 2016. Still, 9 of them picked one that they edited only before. We exclude these responses.
(3) *High deviation:* We also exclude responses where answers to the last two questions deviate by more than 100% (75% considering the lower bound for lines of code) from the real value. In 5 cases this appears for the date of the last edit (measured in days) and in 9 other cases for the lines of code.

As we show in Table 1 (the delta between responded and included), we exclude 18 responses, often due to multiple criteria. Still, 60 responses remain valid and provide the basis for our analysis.

5 RESULTS

In this section, we describe for each research question the results of our survey and discuss the corresponding implications. We display an overview of our data in Table 2.[1] Here, we show all values grouped by the subjective familiarity (SF) that each participant estimated. Furthermore, we show the number of commits that were done by the participant (#C) as well as the number of participants (#P) that responded with this combination of familiarity and commits. We remark, that the commits actually refer to distinct days on which the participant submitted at least one commit. Otherwise, we would, for example, consider multiple small commits of one developer more important than a larger one by another. This way, we aim to neglect this effect. In addition, the number of participants

[1]All responses (anonymous): https://bitbucket.org/Jacob_Krueger/icse-2018-data

Table 2: Subjective file familiarity (SF) assessment compared to the number of commits (#C) and time since the last commit (ΔD). The number of participants (#P) is explicit and describes how often the corresponding combination of SF and #C appears.

SF	1	2			3				4			5			6				
#C	1	1	2	4	1	2	3	6	1	2	7	1	2	9	1	2	4	5	21
#P	4	8	1	1	6	2	1	1	3	1	1	2	1	1	1	1	1	1	1
ΔD	206.3	146.6	317	86	70.8	169	60	184	52	159	41	58	114	25	25	23	28	23	44

SF	7					8									9					
#C	1	3	4	8	27	1	2	5	6	7	9	11	15	27	3	4	16	35	37	43
#P	1	2	1	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
ΔD	10	183	43	91	100	9	15	38	96	115	30	299	137	114	41	234	55	43	34	151

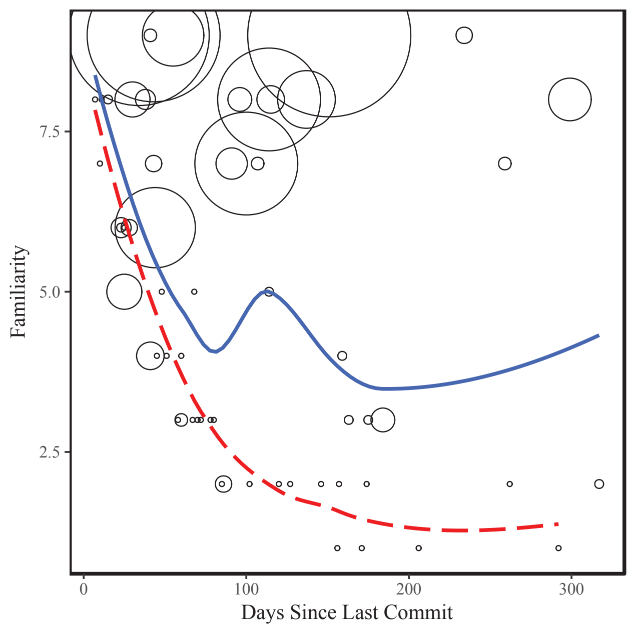

Figure 2: Familiarity of all responses related to the time since the last commit and the number of commits (circle-sizes represent #C). The solid blue line displays average values. The dashed red line displays the average for responses with a single commit.

is explicit (not represented by the number of columns) and we have between 4 and 10 participants for each familiarity level. Finally, we provide an overview on the average days since the last commit (ΔD) based on the date on which the survey has been answered.

First, we consider Figure 2 to describe the necessity for our research questions. Here, we display all responses by relating the days since the last commit to the subjective familiarity. Each circle represents one participant and the circles' sizes the absolute number of commits. In addition, the solid blue line illustrates the average value for all participants, while the dashed red line illustrates the average for those that committed only once.

If we assume that all developers have the same memory strength and that no other factors influence how well they can remember code, the average in Figure 2 should resemble the forgetting curve of Ebbinghaus [15]. At the beginning, this seems to be the case, as most participants state a high familiarity if their last commit is not far in the past. However, around the value of 100 days since the last commit, the average familiarity rises. We see in Figure 2 that the responses with high familiarity at this point skew the curve.

Overall, the average does not follow the forgetting curve. Also considering the peak at around 120 days, this implies two possibilities: Firstly, the forgetting curve is unsuitable for software developers. Secondly, there are other factors besides the time that influence familiarity. As the responses with a single commit (the dashed red line in Figure 2) fit the forgetting curve better, we favor the second option, which means that adaptations to the curve are necessary. This matches our first and third research question, which we investigate in the following.

5.1 Factors' Impact on Familiarity

Regarding our first research question, we aim to identify if there are correlations between the subjective familiarity and the three factors repetition, ratio of own code, and tracking. We support our investigations of each factor with two rank correlation measures: Spearman's Rho (r_s) and Kendall's Tau (τ) [18, 24, 35, 59]. Both are used to asses monotonic dependencies between two variables without assuming normal distribution. The results range from -1 to 1, meaning negative and positive correlation, respectively. We apply for each measure a corresponding significance test with a confidence interval of 0.95 – using algorithm AS 89 [6] for Spearman's Rho and a tau test for Kendall's Tau, as implemented in the statistical programming language R [30].

Before investigating the mentioned factors, we have to test whether the size of a file correlates to familiarity. This should not be the case, as we – in accordance with Ebbinghaus' [15] forgetting curve – consider each file as a single artifact and the remaining familiarity in percentages. The statistics show no significant correlation ($p > 0.2$). For completeness, we still compute the effect sizes, which are very weak for both measures ($r_s = 0.16$ and $\tau = 0.11$). Thus, all results confirm our assumption that the file size does not correlate with the stated familiarity. We summarize the significance tests and correlation measures for all factors (including file size) at the end of this section in Table 3.

5.1.1 Repetition. As first factor that may influence the deviation between the theoretical forgetting curve by Ebbinghaus [15] and the empirical average, we consider repetition. Research shows that repetition can significantly improve memory and learning [45, 56]. In Figure 2, we see that the increase around 100 days since the

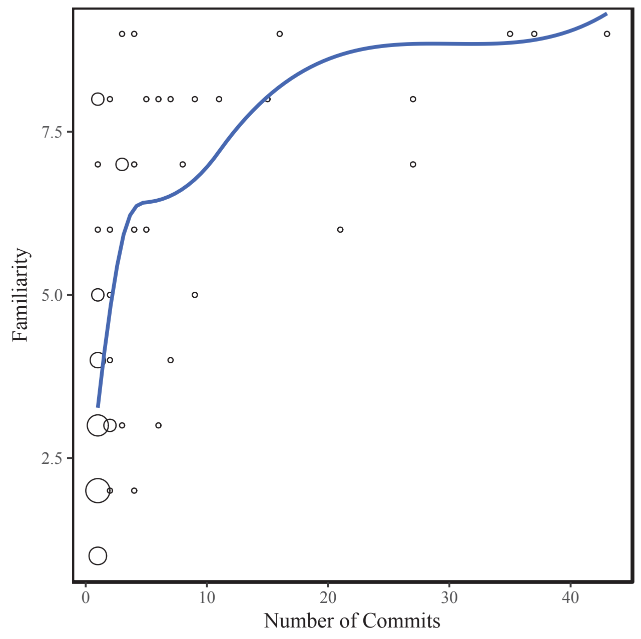

Figure 3: Familiarity related to the number of commits. The blue line displays average values. The circle-sizes represent the number of participants with this combination.

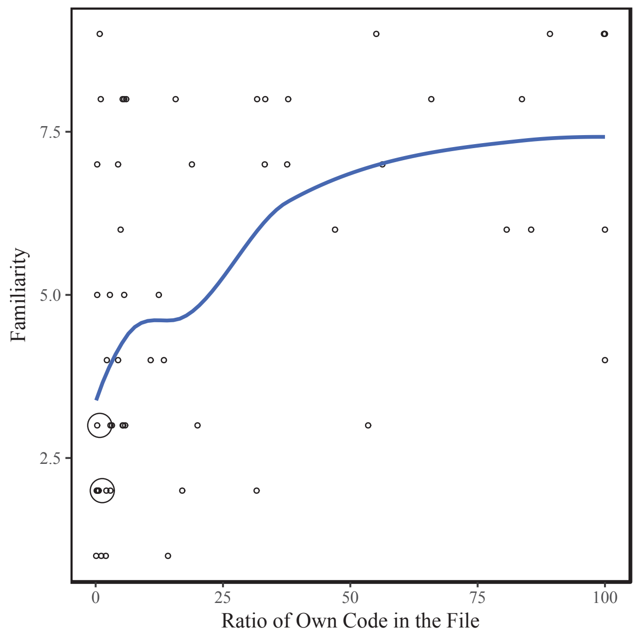

Figure 4: Familiarity related to the ratio of own code. The blue line displays average values. The circle-sizes represent the number of participants with this combination.

last commit matches with a higher number of total commits. Furthermore, we see that most responses with multiple commits are above a familiarity of 5. Responses with fewer – mostly singular – commits are mainly below this threshold.

We can further support this observation by displaying the average only for those responses with a single commit as the dashed red line in Figure 2. In this case, the average actually resembles the forgetting curve of Ebbinghaus [15]. This is reasonable as the curve does not consider repetitions and, thus, may be appropriate for single commits.

In Figure 3, we show the familiarity solely related to the number of commits. Here, the circles' sizes illustrate the number of participants stating this combination. Again, the blue line represents average values. We see that all responses below a familiarity of 5 have 10 or less commits. On average, the results show a rising familiarity as the number of commits increases.

Discussion. All results indicate that repetition affects familiarity. The average deviates from the forgetting curve at points at which responses with multiple commits appear. This is due to these responses being mostly located in the top half of the familiarity scale, where they should not be according to the forgetting curve. In Figure 3, this effect is emphasized even more, as only responses with less than 10 commits in total are below this threshold. To test this, we assume the null hypothesis that commits and familiarity are not correlated. However, our statistics reveal a highly significant correlation between the two ($p < 0.001$). The rank correlation measures imply a moderate to strong positive effect ($r_s = 0.67$ and $\tau = 0.55$). We therefore reject the null hypothesis in favor of assuming that the number of commits positively affects familiarity.

Based on the results we conclude:

> The number of edits is moderately to strongly positively correlated with familiarity in software development.

5.1.2 Ratio of Own Code. Another factor that may impact a developer's familiarity is the ratio of code they implement themselves. To compute this ratio, we extract the file's version for the day we received the survey. Then, we account each line the participants edited last (using `git blame`) to them and relate the sum to the file size. We display the corresponding results in Figure 4. Again, the blue line represents the average and the circle-sizes the number of participants with this combination. The average behaves comparable to the one we find for repetitions (cf. Figure 3). However, the line is on a lower familiarity level in this case.

Discussion. The results imply a correlation between the ratio of own code and familiarity. Still, as the average trend is not as strong as in Figure 3 and more deviation occurs, we assume a weaker correlation. This seems reasonable, as the developer has implemented the code but can only regain familiarity based on repetitions. For our significance tests, we assume as null hypothesis that the ratio of code a developer implemented is not correlated to their familiarity. The outcome indicates a highly significant correlation ($p < 0.001$), wherefore we reject the null hypothesis in favor of assuming that both parameters are related. As rank correlations, we compute $r_s = 0.55$ and $\tau = 0.42$ and, thus, a positive, moderate correlation.

Based on the results we conclude:

> The ratio of code implemented by developers themselves is moderately positively correlated with their familiarity.

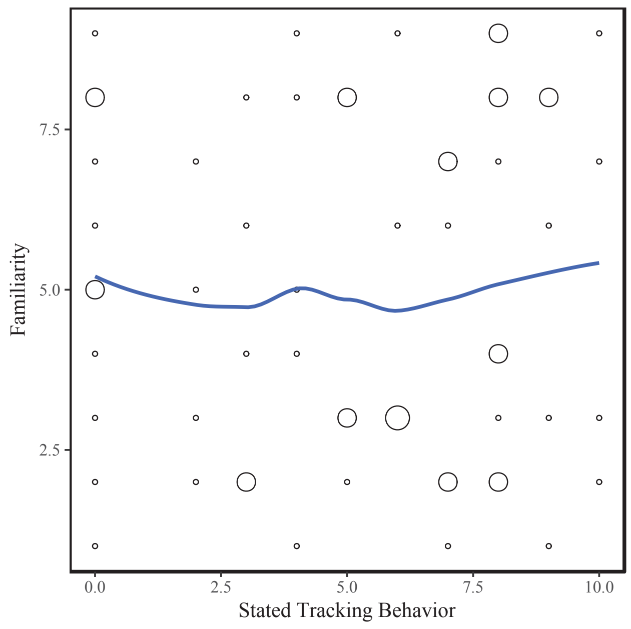

Figure 5: Familiarity related to the tracking behavior. The blue line displays average values. The circle-sizes represent the number of participants with this combination.

5.1.3 Tracking. We asked our participants to which extent they track changes others apply to their code. This factor could also impact their familiarity, because tracking may indicate that they analyze the changes and are therefore more familiar with the code. We display the results in Figure 5. Actually, it surprises us that many of our participants state a value of above 5. This indicates that they are aware of the code others implement and change in their files. Still, participants with high tracking values are almost equally distributed above and below a familiarity of 5.

Discussion. Despite high values in the responses, we find no hint of a correlation between tracking and familiarity in Figure 5. Due to the deviation of responses, we assume no correlation between the two considered parameters. We use this as our null hypothesis and find no significant correlation ($p > 0.78$). Also, the rank correlation measures of $r_s = 0.04$ and $\tau = 0.02$ indicate almost no dependency. Thus, for our participants, we find no correlation between tracking changes and familiarity.

This result poses some questions. Possible explanations may be that developers indeed track changes but do not investigate them. Maybe, our participants also interpreted the term *tracking* differently. For example, some may have seen it as actually analyzing code, but others as just reading notifications. If they actually review code, we would expect a correlation to the familiarity, as these activities are also a form of repetition. However, this does not seem to be the case. Further qualitative analyses are necessary to investigate this discrepancy.

Based on the results we conclude:

The tracking behavior of own files does not affect familiarity.

Table 3: Spearman's Rho (r_s), Kendall's Tau (τ), and the corresponding significance (sig.) values for each factor.

Factor	r_s	sig.	τ	sig.
File Size	0.162	0.218	0.11	0.236
Repetition	0.671	4.557×10^{-9}	0.546	5.175×10^{-8}
Own Code	0.553	4.57×10^{-6}	0.42	6.863×10^{-6}
Tracking	0.036	0.788	0.023	0.81

5.1.4 Summary. Overall, we find that repetitions are positively correlated with familiarity. This is not surprising, as learning and memorizing are improved with repetitions. Interestingly, the number of commits seems to partly outweigh time as an indicator for the subjective familiarity. For the ratio of code implemented by a developer, we also find a positive but weaker correlation. It seems clear that the code developers implement themselves is more familiar to them. Still, they also become unfamiliar with this code, reducing their familiarity if they do not repeatedly investigate it. Considering the tracking of changes, we find no correlation.

Regarding our first research question we conclude:

Repetition as well as the ratio of own code are significantly positively correlated with familiarity. Thus, they must be considered in a suitable forgetting curve.

5.2 Memory Strength

Regarding our second research question, we want to identify an average memory strength for our participants. For this purpose, we compute the memory strength of each participant first by transposing Equation 1 into Equation 2.

$$s = -\frac{t}{ln(R)} \tag{2}$$

Recall that t represents the days since the last commit and R is the stated familiarity. Consequently, s indicates how fast our participants' memory fades each day.

We compute three different distributions: Firstly, the memory strength based on the subjective familiarity of all participants. This value includes repetition and, thus, is biased considering the actual forgetting rate. However, we can again verify our previous findings: If repetition is significant, the median and distribution of the memory strength should be higher than for the other two cases. Secondly, we compute the memory strength based on the subjective familiarity of participants that committed only once. Finally, we compute the memory strength based on the responses to the question after how much time half of a file is forgotten. Here, the retention rate R is 0.5, meaning that half the familiarity is lost, and the time is the participant's response. As this question is challenging to answer, we assume that the second value should represent the best approximation for our participants' memory strength.

Results. We display the computed distributions for each case as violin plots in Figure 6. Note that we use the median instead of the mean for our calculations in the next section, as we have large outliers. These outliers are only partly visualized to avoid an unreadable scaling. We also display the number of responses in each distribution below the corresponding violin plot.

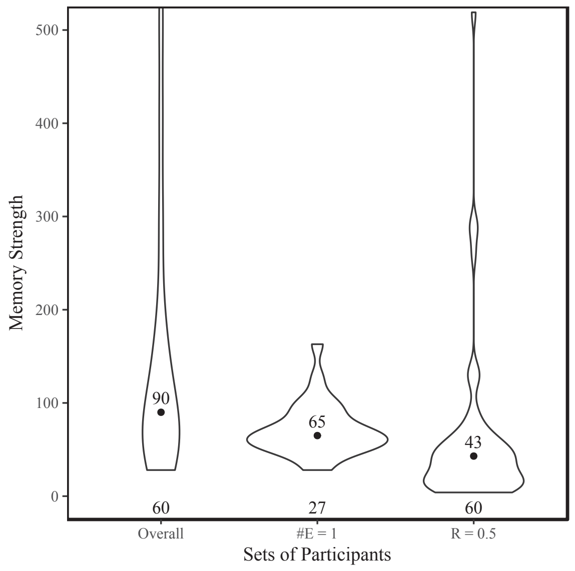

Figure 6: Computed memory strengths based on the familiarity of all 60 participants (Overall, left), of only those 27 with one commit (#E = 1, center), and based on the approximation question (R = 0.5, right).

As we expect, the memory strength based on the familiarity of all participants is the highest with a median of 90. In addition, the deviation in the results is the largest. For the participants with only a single commit (#E = 1) and the approximation question, the medians are 65 and 43, respectively. Both have considerably smaller deviations than the first sample. While the sample with only single commits has the smallest deviation, it also contains only 27 out of 60 responses. Regarding the approximation question, we remark that our participants state that they forget half of their files after 30 days in the median and 40 days on average. To compute the corresponding familiarity, we again use the median.

Discussion. For our goal of deriving a general memory strength, we discard the overall sample as it includes repetition. However, the higher deviation and median substantiate our previous findings. Due to repetition, the computed memory strength increases, indicating the same effects we find in Section 5.1.

Interpreting the remaining two samples is quite difficult. In the sample #E = 1, we compute the values based on the remaining familiarity and only for participants that committed once to a file. Due to the smaller sample size, less deviation appears. The median value of 65 indicates that after this number of days, developers remember only 36.79% of the original file. After approximately 45 days, half of the knowledge is lost.

The median of 43 for the approximation sample (R = 0.5) could represent a more complete view on the participants' familiarity, as we include all of them in this distribution. However, instead of assessing only their subjective familiarity, each participant also has to estimate a time factor. Thus, these results seem less reliable.

This analysis has to be repeated and validated in further studies. Still, we argue that 65 can be considered as a good approximation of the general memory strength for our participants. It is based on less subjective assessments, includes only the appropriate subjects, and is affected by less deviation. In addition, this memory strength does closely correspond to the stated days – on average – after which half of a file is forgotten (40).

Regarding our second research question we conclude:

The computed memory strengths substantiate the previous results on repetition. A median value of 65 seems to be an appropriate approximation of our participants' memory strength.

5.3 The Forgetting Curve

Finally, we consider Ebbinghaus' [15] forgetting curve to answer our third research question. As stated before, if there are no other factors than time, the results we show in Figure 2 should resemble the forgetting curve. If we only consider responses with a single commit (dashed red line), our results and the curve become more similar. Still, our previous results show that there are factors that influence familiarity in software engineering.

In Figure 7, we display the familiarities computed with Equation 1 – based on the previously derived memory strengths – compared to the subjective assessment of our participants. Ideally, one of the curves would resemble equal values for both familiarities. We illustrate this with the black diagonal. As we see, none of the functions resembles this line completely. This is not surprising, as the forgetting curve does not consider any other factor than time. For instance, we find that a high number of commits indicates a high familiarity and, for this reason, all functions drop at a certain point.

Still, as we explained before, Ebbinghaus' [15] curve does roughly resemble the black line if we consider the memory strength of 65 and only single commits. This is indicated by the orange line and triangles being close to the ideal until a familiarity of 6. However, as we determined this memory strength based on the illustrated values, this match is not surprising.

Discussion. The results of our study indicate that the forgetting curve of Ebbinghaus [15] is applicable in software engineering if no other factors than time affect familiarity. In our study, most deviation occurs due to repetition. Consequently, the curve could be used if developers would not modify their code again. Still, this is usually not the case and adapted approaches for software development seem necessary. These can base on our analysis and potentially integrate our findings into an existing forgetting curve, for example by Ebbinghaus [15].

The results we show in Figure 7 also substantiate that the responses of the approximation question seem less reliable. Estimating two subjective values may have negatively influenced the self-assessment of our participants. However, we cannot finally conclude which of the curves represents reality best, as we rely on subjective self-assessments. Thus, further empirical studies are needed to validate and consolidate the memory strength, potentially with different measurements.

Regarding our third research question we conclude:

The forgetting curve of Ebbinghaus [15] is only applicable for software developers if no other factors, mainly repetitions, occur. Thus, an adaptation seems necessary.

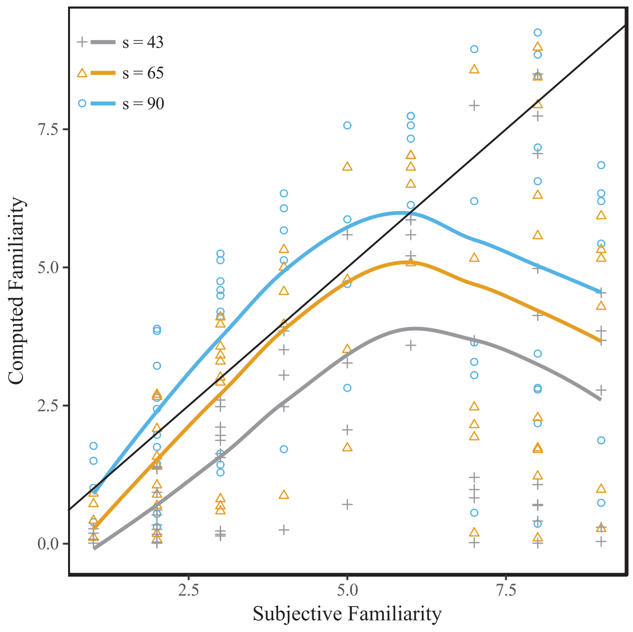

Figure 7: Comparison of subjective and computed familiarities with different memory strengths.

6 THREATS TO VALIDITY

We are aware of several threats to validity, which we discuss based on common classifications [13, 53, 68]. Most threats result from combining two research areas: Psychology and computer science.

Construct Validity. The terms we used may result in misunderstandings and our questions may have been misinterpreted. Especially, this could be a problem regarding that native and non-native English speakers participated. We mitigate this threat by using control questions, as explained in Section 4, and excluding responses that indicate misunderstandings (e.g., proposing a file the participant did not commit to).

Internal Validity. As we aim to measure the familiarity of developers with the source code, we find several threats to the internal validity, due to potentially unknown or not yet considered parameters. In the following, we exemplify some aspects that can affect learning and forgetting, but are excluded from our study:

- The effect of *reviewing* and *testing* but not committing source code is not considered, but also results in repetitions.
- Different *development approaches* may influence how developers remember source code or whether we consider them correctly during our analysis.
- The degree of *reuse* of source code may further support developers' memory, due to multiple occurrences.
- Considering *implemented features*, *development time*, and *importance* of source code could indicate whether developers can remember such factors more easily.
- Some *programming languages* may be harder or easier to remember than others.

Despite such factors, we intentionally used a simplistic approach based on a prominent forgetting curve to gain insight into forgetting in software engineering. While all the aforementioned factors can have an impact, further investigations are necessary to analyze these and, currently, we would have to rely on many assumptions. Also, we argue that considering forgetting, unknown code, and repetition are valid and important factors to this end.

Another threat to the internal validity is the used forgetting curve of Ebbinghaus [15]. Other curves may be better suited to represent forgetting in software engineering and may consider additional parameters. However, the curve we use is established and in a recent study, Murre and Dros [49] replicate and validate its suitability in an experiment. As they also show that other forgetting curves do not heavily differ, we argue that this is not threatening our study. Furthermore, Averell and Heathcote [4] also show that the exponential nature of the forgetting curve fits best to their participants' results. We remark that both studies origin from the psychological domain and, thus, may not be completely transferable.

External Validity. Background factors, such as, age, gender, education, or the motivation of open-source developers [22, 26, 63] – and our respondents in particular – may influence memory performance. However, medical and psychological studies suggest that memory performance is stable until middle age [51] and gender mainly affects episodic memory [25], which is unimportant to our study. We assume that the educational level and motivation are relatively homogeneous in our sample. Still, as we cannot control these factors, they remain a threat to validity.

An additional concern is the subjective nature of familiarity. Each developer learns, understands, and forgets at a different rate, with different factors influencing familiarity. Still, as we rely on an accepted model for forgetting, we argue that by using medians of the participants' results, we obtain valid insights into forgetting in software development.

Conclusion Validity and Reliability. Potentially the main threat to our work are several of our questions requiring subjective self-assessment. This could bias our conclusions in several ways, but as we measure and compute subjective factors, we have to rely on these assessments until we know more about such factors. In addition, we have a comparatively small number of participants, which may lead to statistical errors. We mitigate these threats with our control questions – excluding implausible responses – and by carefully deriving conclusions not only from statistical tests – which we only use to support our arguments. Considering the applied tests, we especially used Spearman's Rho and Kendall's Tau as they do not require normal distributions or linear correlations.

Despite the discussed threats, we argue that any researcher can repeat our study on their own. Depending on the subjects, questions, and parameters, different results may occur. However, this is true for most empirical studies and is not a threat to our study. Nonetheless, we strongly encourage the research community to replicate and extend our approach and study, as we also aim to do. For this purpose, we provide access to an anonymous version of our results, as we described in Section 5.

7 RELATED WORK

There exist several related works that investigate *forgetting* in different domains. Other complementary works include *empirical studies* and *expert identification*.

Forgetting. Nembhard and Osothsilp [50] report a comparative study on forgetting models in the context of production management. They identify several strengths and weaknesses of the models considering different tasks. While this work has a different scope than ours, utilizing the applied method can help to adapt an approach for software development. Also, the problem of correctly approximating forgetting and familiarity can be seen, as few years later Jaber and Sikstrom [32] criticize the aforementioned work. They contradict the results for one model that performs poorly in the previous study. Similarly, Jaber and Bonney [31] compare three learning and forgetting models on a mathematical level.

In psychology, the form of forgetting curves is often debated and Averell and Heathcote [4] address this issue with an experiment. To this end, they measure different variables over 28 days to observe forgetting. The results indicate that exponential forgetting curves, such as the one by Ebbinghaus [15], are the best fit for their participants. Their analysis may provide further details for refining our study and to derive an approach for software engineering.

Empirical Studies. In an empirical study with 19 Java developers, Fritz et al. [19] investigate the identification of knowledge in software development. The participants are asked questions for files they worked regularly or recently on. Both metrics are helpful to identify the experts of a program element and several aspects that can improve the model are investigated. Our study is complementary to this one as we are not interested on identifying existing knowledge, but its fading over time. We also show a significant correlation between regularly working on a file (i.e., repetition) and familiarity, which supports the assumptions of Fritz et al. [19].

Kang and Hahn [34] investigate learning and forgetting in software development. Their findings suggest that learning effects appear for all kind of technology while only methodological knowledge exhibits forgetting. However, they perform their analysis on artificial project data rather than with participants and focus on general categories of knowledge. Our work differs as we examine familiarity on the code level and conduct our study with developers.

LaToza and Myers [40] investigate questions that programmers face while developing software. For this purpose, they gather more than 300 questions and categorize them. The results indicate that developers often ask rather specific questions about a scenario, such as, the impact of a potential bug. Most of these questions are connected to the source code and illustrate the importance of being familiar with it. Thus, their work can be used as basis for extending our study by defining more detailed questions, potentially to approximate familiarity for validation purposes.

Koenemann and Robertson [36] report an empirical study in which they investigate how professional developers analyze source code. Their findings show that programmers only focus on those parts of a software that are relevant to them. Combining these results with ours could imply some further factors that we have to consider when approximating familiarity.

Expertise Identification. Mockus and Herbsleb [48] propose the *Expertise Browser*, a tool to identify experienced developers and experts. To this end, they rely on change management and quantify the changes implemented by a developer as experience. We are not aware of their approach considering that even experts forget and become unfamiliar. Thus, our analysis confirms their assumptions and complements their approach. This also applies to other expertise identification approaches and tools, which focus on communities as well as source code [46, 47, 58].

Of these approaches, the one proposed by Fritz et al. [20] may be the one closest to our study. The authors derive a model to identify experts from previous empirical studies. Here, they consider developers' authorship and interactions with a piece of code to represent their familiarity, which are additional factors that we have to consider. However, while this approach is based on repetition, we are not aware of any consideration of forgetting. Thus, our approach may improve their model by also including this factor.

Anvik et al. [3] describe an approach to assign bug reports based on the previously performed bug fixes of a developer, using machine learning. They use this knowledge to identify the most suitable expert to resolve the new problem. Our insights complement this analysis, as we investigate at which point an expert may have lost too much knowledge.

8 CONCLUSIONS

In this paper, we investigated forgetting in the context of software engineering. For this purpose, we conducted an empirical study with 60 developers. We relied on a simple but valid forgetting curve to analyze their familiarity with a file. With our study, we identify to which extent the original curve represents the subjective assessment of developers. Furthermore, we investigate the importance of three factors on familiarity and derive a representative memory strength for our participants. To conclude our findings, we find:

- The forgetting curve of Ebbinghaus [15] is appropriate in software development if only time has to be considered.
- Repetitions moderately to strongly correlate to familiarity and can be even more important than the elapsed time.
- The ratio of code a developer implemented is moderately correlated to familiarity.
- We need to better understand how developers track their code, as we find no correlation to familiarity.
- A value of 65 for the memory strength seems to be an appropriate value regarding our participants.

We remark that there are several threats to our work and only further studies and research can validate the results. Nonetheless, we do provide important insights into familiarity in the context of software development.

In future work, we will measure familiarity in more detail to investigate forgetting. Currently, we aim to compare subjective and measurable familiarity. Integrating an automated approach based on our results and related works is interesting. Also, additional artifacts of a project and other factors, such as learning, need to be integrated. To this end, additional studies are essential to validate the results and identify further factors that influence familiarity. We see the need for interview studies and action research to derive qualitative insights into forgetting of developers. With large-scale experiments, these findings can be validated and evaluated in more detail than we could do for now. Furthermore, different forgetting curves and their adaptations should be compared and evaluated regarding their applicability for software developers.

Acknowledgments This research is supported by DFG grants LE 3382/2-1, SA 465/49-1, and Volkswagen Financial Services AG.

REFERENCES

[1] Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim. 2003. The Software Maintenance Project Effort Estimation Model Based on Function Points. *Journal of Software: Evolution and Process* 15, 2 (2003), 71–85.

[2] Nicolas Anquetil, Káthia M de Oliveira, Kleiber D de Sousa, and Márcio G Batista Dias. 2007. Software Maintenance Seen as a Knowledge Management Issue. *Information and Software Technology* 49, 5 (2007), 515–529.

[3] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who Should Fix this Bug? In *International Conference on Software Engineering*. ACM, 361–370.

[4] Lee Averell and Andrew Heathcote. 2011. The Form of the Forgetting Curve and the Fate of Memories. *Journal of Mathematical Psychology* 55, 1 (2011), 25–35.

[5] Jongmoon Baik, Barry W Boehm, and Bert M Steece. 2002. Disaggregating and Calibrating the CASE Tool Variable in COCOMO II. *IEEE Transactions on Software Engineering* 28, 11 (2002), 1009–1022.

[6] D J Best and D E Roberts. 1975. Algorithm AS 89: The Upper Tail Probabilities of Spearman's Rho. *Journal of the Royal Statistical Society* 24, 3 (1975), 377–379.

[7] Barry W Boehm. 1976. Software Engineering. *IEEE Transactions on Computers* C-25, 12 (1976), 1226–1241.

[8] Barry W Boehm. 1981. *Software Engineering Economics*. Prentice-Hall.

[9] Barry W Boehm, Chris Abts, and Sunita Chulani. 2000. Software Development Cost Estimation Approaches - A Survey. *Annals of Software Engineering* 10, 1 (2000), 177–205.

[10] Barry W Boehm, Chris Abts, Bradford K Clark, Ellis Horowitz, A Winsor Brown, Donald Reifer, Sunita Chulani, Ray Madachy, and Bert Steece. 2000. *Software Cost Estimation with COCOMO II*. Prentice Hall.

[11] Barry W Boehm, A Winsor Brown, Ray Madachy, and Ye Yang. 2004. A Software Product Line Life Cycle Cost Estimation Model. In *International Symposium on Empirical Software Engineering*. IEEE, 156–164.

[12] Elliot J Chikofsky and James H Cross. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* 7, 1 (1990), 13–17.

[13] Thomas D Cook and Donald Thomas Campbell. 1979. *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin.

[14] Eric L Dey. 1997. Working with Low Survey Response Rates: The Efficacy of Weighting Adjustments. *Research in Higher Education* 38, 2 (1997), 215–227.

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

[16] J Alberto Espinosa, Sandra A Slaughter, Robert E Kraut, and James D Herbsleb. 2007. Familiarity, Complexity, and Team Performance in Geographically Distributed Software Development. *Organization Science* 18, 4 (2007), 613–630.

[17] Jean-Marie Favre, Jacky Estublier, and Remy Sanlaville. 2003. Tool Adoption Issues in a Very Large Software Company. In *International Workshop on Adoption-Centric Software Engineering*. Carnegie Mellon University, 81–89.

[18] Gregory A Fredricks and Roger B Nelsen. 2007. On the Relationship Between Spearman's Rho and Kendall's Tau for Pairs of Continuous Random Variables. *Journal of Statistical Planning and Inference* 137, 7 (2007), 2143–2150.

[19] Thomas Fritz, Gail C Murphy, and Emily Hill. 2007. Does a Programmer's Activity Indicate Knowledge of Code? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 341–350.

[20] Thomas Fritz, Jingwen Ou, Gail C Murphy, and Emerson Murphy-Hill. 2010. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *International Conference on Software Engineering*. ACM, 385–394.

[21] Arthur M Glenberg. 1976. Monotonic and Nonmonotonic Lag Effects in Paired-Associate and Recognition Memory Paradigms. *Journal of Verbal Learning and Verbal Behavior* 15, 1 (1976), 1–16.

[22] Alexander Hars and Shaosong Ou. 2001. Working for Free? Motivations of Participating in Open Source Projects. In *Hawaii International Conference on System Sciences*. IEEE, 1–9.

[23] Les Hatton. 1998. Does OO Sync with How We Think? *IEEE Software* 15, 3 (1998), 46–54.

[24] Jan Hauke and Tomasz Kossowski. 2011. Comparison of Values of Pearson's and Spearman's Correlation Coefficients on the Same Sets of Data. *Quaestiones Geographicae* 30, 2 (2011), 87–93.

[25] Agneta Herlitz, Lars-Göran Nilsson, and Lars Bäckman. 1997. Gender Differences in Episodic Memory. *Memory & Cognition* 25, 6 (1997), 801–811.

[26] Guido Hertel, Sven Niedner, and Stefanie Herrmann. 2003. Motivation of Software Developers in Open Source Projects: An Internet-Based Survey of Contributors to the Linux Kernel. *Research Policy* 32, 7 (2003), 1159–1177.

[27] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. 2017. Shorter Identifier Names Take Longer to Comprehend. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 217–227.

[28] Robert S Huckman and Bradley R Staats. 2011. Fluid Tasks and Fluid Teams: The Impact of Diversity in Experience and Team Familiarity on Team Performance. *Manufacturing & Service Operations Management* 13, 3 (2011), 310–328.

[29] Robert S Huckman, Bradley R Staats, and David M Upton. 2009. Team Familiarity, Role Experience, and Performance: Evidence from Indian Software Services. *Management Science* 55, 1 (2009), 85–100.

[30] Ross Ihaka and Robert Gentleman. 1996. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* 5, 3 (1996), 299–314.

[31] Mohamad Y Jaber and Maurice Bonney. 1997. A Comparative Study of Learning Curves with Forgetting. *Applied Mathematical Modelling* 21, 8 (1997), 523–531.

[32] Mohamad Y Jaber and S Sikstrom. 2004. A Note on "An Empirical Comparison of Forgetting Models". *IEEE Transactions on Engineering Management* 51, 2 (2004), 233–234.

[33] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *International Systems and Software Product Line Conference*. ACM, 61–70.

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

[35] Maurice G Kendall. 1938. A New Measure of Rank Correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[36] Jürgen Koenemann and Scott P Robertson. 1991. Expert Problem Solving Strategies for Program Comprehension. In *Conference on Human Factors in Computing Systems*. ACM, 125–130.

[37] Jacob Krüger, Stephan Dassow, Karl-Albert Bebber, and Thomas Leich. 2017. Daedalus or Icarus? Experiences on Follow-the-Sun. In *International Conference on Global Software Engineering*. IEEE, 31–35.

[38] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference*. ACM, 354–361.

[39] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 105–112.

[40] Thomas D LaToza and Brad A Myers. 2010. Hard-To-Answer Questions About Code. In *Evaluation and Usability of Programming Languages and Tools*. ACM, 8.

[41] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering* 3, 4 (2007), 303–318.

[42] Sherlock A Licorish and Stephen G MacDonell. 2014. Understanding the Attitudes, Knowledge Sharing Behaviors and Task Performance of Core Developers: A Longitudinal Study. *Information and Software Technology* 56, 12 (2014), 1578–1596.

[43] Glenn Littlepage, William Robison, and Kelly Reddington. 1997. Effects of Task Experience and Group Experience on Group Performance, Member Ability, and Recognition of Expertise. *Organizational Behavior and Human Decision Processes* 69, 2 (1997), 133–147.

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

[45] Richard E Mayer. 1983. Can You Repeat That? Qualitative Effects of Repetition and Advance Organizers on Learning from Science Prose. *Journal of Educational Psychology* 75, 1 (1983), 40–49.

[46] David W McDonald and Mark S Ackerman. 2000. Expertise Recommender: A Flexible Recommendation System and Architecture. In *Conference on Computer Supported Cooperative Work*. ACM, 231–240.

[47] Shawn Minto and Gail C Murphy. 2007. Recommending Emergent Teams. In *International Workshop on Mining Software Repositories*. IEEE.

[48] Audris Mockus and James D Herbsleb. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *International Conference on Software Engineering*. ACM, 503–512.

[49] Jaap M J Murre and Joeri Dros. 2015. Replication and Analysis of Ebbinghaus' Forgetting Curve. *PLoS ONE* 10, 7 (2015), 1–23.

[50] David A Nembhard and Napassavong Osothsilp. 2001. An Empirical Comparison of Forgetting Models. *IEEE Transactions on Engineering Management* 48, 3 (2001), 283–291.

[51] Lars-Göran Nilsson. 2003. Memory Function in Normal Aging. *Acta Neurologica Scandinavica* 107 (2003), 7–13.

[52] Gerardo Andrés Okhuysen. 2001. Structuring Change: Familiarity and Formal Interventions in Problem-Solving Groups. *Academy of Management Journal* 44, 4 (2001), 794–808.

[53] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. 2000. Empirical Studies of Software Engineering: A Roadmap. In *Conference on The Future of Software Engineering*. ACM, 345–355.

[54] Andreas Riege. 2005. Three-Dozen Knowledge-Sharing Barriers Managers Must Consider. *Journal of Knowledge Management* 9, 3 (2005), 18–35.

[55] Martin P Robillard, Wesley Coelho, and Gail C Murphy. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Transactions on Software Engineering* 30, 12 (2004), 889–903.

[56] Irvin Rock. 1957. The Role of Repetition in Associative Learning. *The American Journal of Psychology* 70, 2 (1957), 186–193.

[57] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending Studies on Program Comprehension. In *International Conference on Program Comprehension*. IEEE, 308–311.

[58] David Schuler and Thomas Zimmermann. 2008. Mining Usage Expertise from Version Archives. In *International Working Conference on Mining Software Repositories*. ACM, 121–124.

[59] Pranab K Sen. 1968. Estimates of the Regression Coefficient Based on Kendall's Tau. *Journal of the American Statistical Association* 63, 324 (1968), 1379–1389.

[60] Teresa M Shaft and Iris Vessey. 2006. The Role of Cognitive Fit in the Relationship Between Software Comprehension and Modification. *Management Information Systems Quarterly* 30, 1 (2006), 29–55.

[61] David Sharon. 1996. Meeting the Challenge of Software Maintenance. *IEEE Software* 13, 1 (1996), 122–125.

[62] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An Examination of Software Engineering Work Practices. In *CASCON First Decade High Impact Papers*. IBM, 174–188.

[63] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution*. IEEE, 151–160.

[64] Thomas A Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* 5 (1984), 494–497.

[65] Armstrong A Takang, Penny A Grubb, and Robert D Macredie. 1996. The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation. *Journal of Programming Languages* 4, 3 (1996), 143–167.

[66] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. 2007. Empirical Studies in Reverse Engineering: State of the Art and Future Rrends. *Empirical Software Engineering* 12, 5 (2007), 551–571.

[67] Bart Van Den Hooff and Jan A De Ridder. 2004. Knowledge Sharing in Context: The Influence of Organizational Commitment, Communication Climate and CMC use on Knowledge Sharing. *Journal of Knowledge Management* 8, 6 (2004), 117–130.

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