University of Magdeburg

School of Computer Science

Master's Thesis

Measuring Code Familiarity in Forked Product Variants

Author:

Jens Wiemann

March 30, 2017

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Dipl.-Inf. Wolfram Fenske

M. Sc. Jacob Krüger

Otto-von-Guericke-University Magdeburg
Department of Technical and Business Information Systems

Prof. Dr.-Ing. Thomas Leich

METOP GmbH

Affiliated Institute to the Otto-von-Guericke-University Magdeburg

Wiemann, Jens:

Measuring Code Familiarity in Forked Product Variants
Master's Thesis, University of Magdeburg, 2017.

Abstract

The number of legacy systems that need to modernized rises significantly. At the same time, companies are more and more forced to customize their software products for different customers. Software product lines promise the solution by supplying a high degree of customization, maintenance and extensibility. Before a company can migrate their existing products into a product line, they must evaluate the additional development costs in contrast to the savings gained for future products. Cost estimation models are often used to calculate these costs and whether it is more effective to start from scratch. Besides other factors, these models need to know how familiar the development team is with its software to approximate how much effort needs to be invested to migrate the legacy systems. There are different approaches that use the existing project's version control system to identify the familiarity (expertise) in a project, but these don't regard the memory retention that occurs naturally over time. This is especially important when migrating legacy systems, as the lack of knowledge can be expensive to recover.

In this thesis, we introduce an automated familiarity estimation approach, that is based on the natural memory retention model, which is well established among the psychological community. Our algorithm can calculate how familiar a developer is with a file he wrote, considering the past time after each edit. Based on this concept, we can calculate the developer's familiarity for the entire project. Our approach was tested by conducting a survey that questioned developers for their experience. In our evaluation, we could confirm that our approach can be used to calculate the project familiarity of a developer.

Acknowledgements

I would like to thank my advisors Prof. Gunter Saake, Prof. Thomas Leich, Dipl.-Inf. Wolfram Fenske and M.Sc. Jacob Krüger for their support during this thesis. Their constructive input helped me to considerably improve the quality of content and writing.

Finally, I would like to thank my family and friends for encouragement and moral support.

Contents

List of Figures xi
List of Tables xiii
List of Abbreviations xv
1. Introduction 1
1.1 Contribution 2
1.2 Outline 3
2. Background 5
2.1 Software Migration 5
2.1.1 Software Product Lines 6
2.1.2 Software Reengineering 7
2.1.3 Software Cost Estimation 8
2.2 Version Control Systems 8
2.2.1 Git Blame 9
2.3 Familiarity 10
3. Concept 11
3.1 Requirements 11
3.2 Forgetting Curve 12
3.3 Familiarity Approximation 15
3.3.1 File Familiarity 17
3.3.2 Project Familiarity 20
3.4 Summary 21
4. Experimental Design 23
4.1 Research Questions 24
4.2 Survey Setup 25
4.2.1 Survey Questions 26
4.3 Analysis Tool 29
4.4 Summary 31

5. Evaluation 33
5.1 Survey Execution 33
5.2 Results 35
5.2.1 Familiarity and the Number of Commits 35
5.2.2 General Memory Strength for Software Developers 36
5.2.3 Project Familiarity 40
5.3 Discussion 42
5.4 Threats to Validity 44
5.5 Summary 45
6. Related Work 47
7. Conclusion 49
7.1 Contributions 49
7.2 Future Work 50
A. Appendix 51
A.1 Survey Results 51
Bibliography 55

List of Figures

Figure 1: Forgetting curve with different memory strengths s = 1, 2, 3. 13
Figure 2: Forgetting curve with two reviewing phases. 14
Figure 3: Developer - Commit relationship. 16
Figure 4: Developer – File relationship. 17
Figure 5: File familiarity example with two authors at day 3. 19
Figure 6: Number of edits in relation to the stated familiarity. 36
Figure 7: File familiarities stated in the survey. 37
Figure 8: Boxplots of the unfiltered and filtered memory strengths. 38
Figure 9: Days until half of the knowledge is forgotten. 39
Figure 10: Filtered memory strengths besides the self-assessed memory strengths. 40
Figure 11: Project familiarities stated in the survey. 41
Figure 12: Calculated Project Familiarity compared to Stated Project Familiarity. 42

List of Tables

Table 1: Rating scale for programmer unfamiliarity as used in COCOMO II 12
Table 2: Usage of the queried git blame output.....30
Table 3: Selected projects, the used languages, and currently active developers 34
Table 4: Exclusion criteria and number of occurrences.....34

List of Abbreviations

SPL	Software product line
VCS	Version control systems
ROI	Return on investments
BEP	Break-even point
LOC	Lines of code
UNFM	Programmer unfamiliarity
WPF	Windows Presentation Foundation
API	Application programming interface
PCC	Pearson correlation coefficient

1. Introduction

As competition in the software development domain increases, customers demand highly customized software. In addition, users get accustomed to more and more convenience features, which they expect to get for free, as everybody seems to have them. To cope with this trend, software developers adopted *software product lines (SPL)*, a method to systematically manage and reuse features for similar products [Saake, et al., 2013]. Besides mass customization, software product lines facilitate maintenance, increase product quality, and reduce development effort for new products [Pohl, et al., 2005].

Creating new products for a software product line results in overhead for planning and development, which is why companies rather develop quick and dirty to learn from their early adopters' feedback [Highsmith, 2002]. Many companies start with a single innovative product or are not aware of software product lines. As they want to reuse parts of their product for new customers, companies use unsystematic approaches. This may work for a few variants, but as their number increases, so does their maintenance effort [Pohl, et al., 2005]. At some point, these companies may see the need for a more systematic reuse strategy and must reengineer their legacy systems into a software product line. This is called the extractive approach, and is more common than developing an SPL from scratch (proactive), but requires investments as most of the existing code must be refactored.

One of the unsystematic reuse approaches is copying all the parts needed and modifying them as necessary. This is called *clone-and-own.* It takes economic advantage from previous work, but creates an entirely new system that must be maintained [Fischer, et al., 2014] [Martinez, et al., 2015]. This approach is often used in combination with *version control systems (VCS)*, such as *Git*[1] or *Apache Subversion (SVN)*[2]. Version control systems are an essential development tools to organize and control revisions for multi-

[1] Git: https://git-scm.com [09.11.2016]
[2] SVN: http://subversion.apache.org [09.11.2016]

developer projects. Their branching or forking feature creates a copy of a project at a specific point in time, so that several copies may develop independently from each other. This feature is often used by companies to apply the clone-and-own approach [Rubin, et al., 2012]. As version control systems handle clones independent from each other, changes must be propagated manually to each clone. The manual work is error-prone and costs unnecessary time.

Before introducing a software-product-line approach, a company wants to estimate the costs and potential savings, as the paradigm change provides no immediate benefits. There are two common cost estimation approaches, algorithmic models or expert judgement [Boehm, et al., 1995]. Both methods have their disadvantages, as experts are expensive and algorithmic models require in depth knowledge of the project to determine their required parameters.

The extractive approach to build a software product line is commonly used, as a rising number of legacy systems drastically increases the price for maintenance. In addition, this approach can be done one legacy system at a time, reducing the time to market. Although an SPL promises a high *return on investments (ROI)*, the *break-even point (BEP)* may take several new products after the extraction. This calls for an in-depth analysis before investing in an SPL. Algorithmic cost models can be used to predict the break-even point, and there are some, like the adapted SIMPLE algorithm [Krüger, 2016], that focus on the extractive approach. Nevertheless, they still require expertise in the project to estimate their parameters.

An important part of most cost models, is the estimation on how familiar the team is with its project. Especially in extraction scenarios, the company wants to know how easily their developers can reengineer their old projects. To estimate this, a company must estimate the project familiarity, mostly done by asking the team leaders. In cases of older projects, this can be difficult, as developers often change projects and there is no standardized way to estimate the familiarity. The result are manual guesses that can hardly be compared between projects or companies.

1.1 Contribution

As version control systems track all code changes, they provide in-depth information of the emergence of each product. In combination with the metadata stored with each check-in, for instance author, date, or commit messages, it might provide enough information to automatically calculate the project familiarity. However, existing approaches on determining the familiarity (expertise) of developers by mining version control systems like the Expertise Browser [Mockus, et al., 2002] don't consider the retention of knowledge that naturally occurs over time. This memory retention seems critical, when evaluating whether to reengineer an old project or start by scratch.

Goal of This Thesis

Our goal for this thesis, is to introduce an automated approach on calculating the project familiarity for a team of developers. Thus, we start by introducing an algorithm that calculates how familiar a developer is with a single file and aggregate the results to approximate a familiarity for the entire project. Per Girba et al. [Gîrba, et al., 2005], the fine-granularity of a line basis, rather than calculating the familiarity on a file basis results in more accurate estimations. Thus, we want to calculate the familiarity for every commit, as it is the smallest unit the version control system can provide information of.

Methodology

To achieve our goal, we apply the following methodology. We start by describing the requirements that our algorithm must fulfill. Based on our requirements, we will introduce an algorithm that calculates the familiarity for a single file. Then, we suggest an approach to aggregate each familiarity to obtain the value on how familiar a developer is with the entire project. By, once again, aggregating each developers project familiarity, we obtain the desired familiarity for the entire team.

We conduct a survey to evaluate our results. The survey targets individual developers, which are asked to state their familiarity for a file they worked on. By finding said file and using our algorithm to calculate the familiarity, we can compare our result to the familiarity the participant stated.

Results

During the discussion of our results, we show that our algorithm is suited to estimate the familiarity of a developer regarding a single file and the project. In addition, we found a value for a general developer's memory strength. This value indicates over how much time his memory of a file is forgotten. However, we were not able to prove that our algorithm may be aggregated to estimate an entire teams project familiarity.

1.2 Outline

The structure of this thesis is as follows.

In Chapter 2 we introduce the necessary background information to understand our work. We describe the process of software migration, including software product lines, reengineering and cost estimation. Next, we provide an overview of the features of a version control system including the Git Blame feature which we will use in our work. Finally, we will provide a definition of software familiarity.

Chapter 3 describes the concept of our approach. We start by defining our requirements and introducing the forgetting curve on which our algorithm is based on. Then, we introduce our algorithm that calculates the familiarity of a developer for a single

file. In addition, we explain our approach on aggregating the file familiarities to obtain a project familiarity of a developer and team.

In Chapter 4 we formulate our research questions that must be answered to validate our algorithm. Based on those, we design a survey targeting multiple developers from public open source projects. We then introduce a tool that helps us evaluate the results of the survey.

We begin Chapter 5 with the statistics of the survey execution, followed by the presentation of the results. The results are divided into three parts, each associated to one of the research questions. In the first part, we analyze the relation between the number of edits and the stated familiarity. The second part uses the answers from the survey to calculate an average developer's memory strength. This value is used in the third part to calculate our participants project familiarity which will be then compared to the project familiarity he stated in the survey. Next, we discuss our results and statistical deviations. Finally, we discuss identified threats to validity.

In Chapter 6 we discuss related work and what differentiates our approach from the previous ones.

We conclude this thesis in Chapter 7 and present topics for further research.

2. Background

In this chapter, we introduce the basic concepts that are necessary to understand this thesis. We begin with an introduction to software migration with a focus on software product lines and the reengineering that is often associated with it. An overview of cost estimation for software migration gives insights on the extent of the existing cost estimation models.

As we use version control systems to retrieve our data, we will give the necessary overview to understand the features we use in this thesis. Finally, we present a definition for software familiarity on which we rely on.

2.1 Software Migration

Software migration describes the transform of a software system to a target environment. The migration process is a technical transformation in which the resulting system obtains the original business logic [Gimnich, et al., 2005]. The legacy systems' functionality can be tested after the migration by conducting regression tests [Chen, et al., 1997]. These ensure, that the migration did not compromise the original software requirements.

The migration process is often triggered when the requirements change or the systems interacting with it are no longer compatible. Over the years, legacy systems have grown drastically in numbers. If a modernization seems imminent, there are multiple techniques that should be evaluated.

The simplest technique is to re-host the application. By simply scaling the used hardware or hosting it on a different platform, the application gains the required responsiveness or speed. Unfortunately, this solution does not improve its effectiveness, often postponing a greater migration.

Another option is to migrate the software to a new language, database, or operating system. This often requires in depth changes to the software which makes it expensive in contrast to simple re-hosting. On the upside, this increases the efficiency of the system as operations benefit from the newer underlying soft- and hardware. If the new requirements require the development of new features or extensions, this method is not optimal, as the new code must be built on top of the old software. This results in high maintenance effort as the number of features increases.

The most efficient and agile way of transforming legacy applications is to reengineer them. This method enables the legacy application to build upon new technologies or platforms. In addition, a more effective software design may be applied to facilitate future maintenance or extensions [Menychtas, et al., 2013]. For companies that can predict their growth and realize their need for reusable and extendable software, this method can prepare their software architecture for future changes.

2.1.1 Software Product Lines

A software product line describes a set of similar systems, that are developed from and share common assets [Saake, et al., 2013]. Assets are reusable artifacts that describe and implement a feature [Clements, et al., 2007]. The assets are developed to be used in multiple systems, fulfilling the requirements of a single feature. In this way, a feature is implemented once and used in multiple systems that require this feature. This facilitates future maintenance, extensibility and customizable products [Northrop, 2002]. The optimal software product line requires only the selection of the desired features, which will then generate a functional application with the selected features. Even dependencies between features are considered in this approach [Kastner, et al., 2007].

Software product lines should be used when migrating legacy systems, if the company has multiple applications that share similar features. The resulting set of features is then used to generate variants that replace the legacy applications. New features or changes to existing ones can be used across all new applications with hardly any extra integration effort. If a new variant is required, the software product line can be used to generate it based on the required features. If a new feature is required, it can be implemented without influencing the existing products.

More and more customers expect products that are custom tailored for their needs, resulting in an increased importance for customization [Pohl, et al., 2005]. There are some alternative approaches on delivering this customization, for example the clone-and-own approach. This method is often the first choice for project developers in this domain, as it is easy to implement. The existing project is simply cloned and adapted to fulfill the customers need. This results in great difficulty for maintenance, which increases for each further variant. The cleaner way to migrate legacy systems in this case is to use software product lines.

There are three approaches defined by Krueger [Krueger, 2002] that companies must evaluate on migrating to software product lines:

- **Proactive** – A new product line is designed and implemented from scratch, migrating only the feature requirements.
- **Reactive** – A new product line is designed with only a subset of all future features. As time passes, more and more features are implemented as the requirements for new variants emerge.
- **Extractive** – Similar and unique features are identified from existing legacy systems. These are reengineered into reusable assets and used in the new product line.

The reactive approach is the most effective approach to introduce a software product line [Clements, 2002]. However, if a company wants to reuse parts of their legacy system, they will use the extractive approach. This is most commonly used by companies, when migrating legacy systems [Pohl, et al., 2005].

2.1.2 Software Reengineering

Software reengineering includes all activities that have the goal of improving the quality of an existing software system [Bergey, et al., 1996]. Most of the time, these improvements aim at prolonging the lifespan of a software. Reengineering implies every transformation that is conducted on the source code, documentation, or associated programs. There are multiple use cases that require reengineering, among others:

- **Maintenance** – Correction of errors.
- **Extension or Adaptation** – Adaptation to changed requirements.
- **Migration** – Transferring the application to a different target environment.
- **Integration** – Combination of different applications.
- **Renovation** – Improvements on the application quality.
- **Re-Documentation** – Improving or completing the documentation.

The presented use cases all require different methods or technical implementations. Nevertheless, they all benefit from or even require in depth knowledge of the source code. Reengineering gets more and more important, as we evolve from writing applications from scratch that already exist in a slightly different way towards evolving existing ones [Winter, 2004].

2.1.3 Software Cost Estimation

Software cost models are used to predict the costs to develop or enhance a software system. They compare the benefits, risks and costs that result from the development. The results of the cost estimations are important to evaluate whether a company should use a software-product-line, or develop a stand-alone application [Dubinsky, et al., 2013].

According to Boehm [Boehm, et al., 1995], there are the following approaches on estimating the costs for a software system:

- **Top-down** – The total project cost is estimated and distributed among individual components.
- **Bottom-up** – The costs for individual components are estimated and aggregated to calculate the overall cost.
- **Analogies** – The costs of a similar completed project are used to estimate the costs.
- **Expert judgement** – An expert's judgement, based on experience, is used to estimate the costs.
- **Algorithmic models** – An algorithmic model is used to calculate the costs based on parameters that represent cost drivers.

They differ so drastically from another, that each of them has its own advantages and disadvantages. All methods should be used if possible to acquire a more complete overview of the costs [Jorgensen, et al., 2009].

Algorithmic models, such as COPLIMO (Constructive Product Line Investment Model) [Boehm, et al., 2004] can be used to evaluate software-product-line engineering. They are often adapted and combined to improve their efficiency for different domains. However, most of them use the same underlying cost drivers and metrics [Krüger, 2016]. One of those metrics is the familiarity of the team with its project.

The programmers' familiarity is used to estimate the amount of effort required to modify the existing software [Boehm, et al., 2004]. A lack of familiarity requires high investments in time and money. This makes it important for any cost estimation.

2.2 Version Control Systems

Version control systems (VCS) are widely used among companies to revision and track their source code over time. They can be used on premise or as web services, such as GitHub. With every change that is committed to the project monitored by the VCS, the system stores information of the submission, such as the time, commit message, author and exact revision it was committed to [Mercurial, 2009]. This information is then

used by the system to merge the user's changes with the project. In the case that multiple changes are committed to the same state of the project, occurring conflicts must be resolved by the user.

VCS are very popular for mining information on projects, as there are many projects that are being hosted on publicly accessible web service based version control systems [Bird, et al., 2009]. These projects are mostly open source, which makes it popular for case studies that require project information. The amount of information provided by these VCS give insights of the entire development of a project.

Distributed version control systems enable developers to contribute code despite their physical location. This enables projects without a conventional team, a team that consists of individual and distributed developers. In addition, the open source aspect invites developers that are not core-developers of the project. This enables a whole new workflow, as voluntary developers contribute to projects that are core-developed and managed by a company [Rodríguez-Bustos, et al., 2012].

Version control systems also allow an easy cloning of a project. This makes it attractive for the clone-and-own approach [Dalgarno, et al., 2007]. The cloned projects are developed individually from another. Therefore, the disadvantage of having to deal with change propagation one version at a time is still given.

2.2.1 Git Blame

To easily identify the developer behind a certain line of code, Git introduced the "Git Blame" feature. This feature shows what revision and author last modified each line of a file [Git, 2017]. Each line of code is associated to a commit, from which the blame feature gathers its information. The following information is provided by the git blame command:

- GUID
- Number of lines removed
- Number of lines moved
- Number of lines added
- Author
- Author time
- Committer
- Committer time
- Summary
- Filename
- Line of code
- Previous commit GUID

A format that is designed for machine consumption can be achieved by adding the parameter '--porcelain'. This command requires the command line interface of git, which can be acquired from the official git project.

2.3 Familiarity

The concept of familiarity in organizational teams has been defined as “the knowledge that members of a team have about the unique aspects of their work” [Goodman, et al., 1988]. Besides the knowledge of the task, the knowledge about other members of the team is also contributed to the familiarity [Littlepage, et al., 1997]. As members of a team work together over time, they become familiar with the domain and the team members [Katz, 1982]. The team members also develop a common knowledge base that improves interaction and the localization of expertise among its members [Littlepage, et al., 1997]. Studies have shown the positive influence of familiarity on team performance in flight simulation [Kanki, et al., 1989], problem solving [Littlepage, et al., 1997] and various other tasks [Harrison, et al., 2003].

The conditions of familiarity have been dramatically changed over the ages by the invention of better ways to store information [Luhmann, 2000]. For example, nowadays we have access to huge amounts of data for a single topic, which one may never be totally familiar with. People in the 16th century in return, could learn every information there existed of a certain topic, as there was rarely any compared to today. In conclusion, the conditions of familiarity must be defined to compare stated familiarities.

3. Concept

In this chapter, we will introduce our algorithm to evaluate the project familiarity for each contributing developer. Firstly, we introduce the forgetting curve, which describes the decline of memory retention in time. This concept, in combination with the information a VCS provides, allows us to estimate how familiar a developer is with files he wrote. Derived from all the project's developers file familiarity, we can estimate the familiarity coverage of the entire project.

3.1 Requirements

We have identified multiple criteria that need to be considered in the approach to analyze a team's project-familiarity. These requirements are explained in the following sections.

- **Automated Analysis** – The current approach on project familiarity is to ask the team members and team leader, which bears the risk of deviating interpretations among the asked developers. Another problem is, that although the same algorithm is used across different teams or companies, their interpretation will vary as well, making it hard to compare the results. The most appealing way to eliminate the questioning of the developers is to automate the entire process. By using an automated algorithm that determines the familiarity, we can compare different projects and teams.
- **Reengineering Support** – As the cost models we are looking at are used to approximate the costs for refactoring and reengineering of former projects, this makes it a major use case. One of the requirements that support the reengineering of a project is, that the developer in charge is familiar with the project. The developer greatly benefits from detailed knowledge of the project, avoiding unfamiliar inheritances, reflections or references. To support this requirement, we need to consider the forgetting of said details over time.

- **Language Independent** – To support a variety of projects and companies, we aim towards a programming language independent algorithm. This will further increase the comparability between analyzed projects.

3.2 Forgetting Curve

An important part of software development is how familiar the project's developers are with the software. This can be seen in current cost estimation models, in which software unfamiliarity plays an important role. One of the most commonly used cost estimation models is COCOMO II, introduced by Boehm in 1995 [Boehm, et al., 1995]. Based on the relative *programmer unfamiliarity (UNFM)*, ranging from 0.0 to 1.0, the effort for future assessment and assimilation of the analyzed project can be estimated [USC: Center for Software Engineering, 2000]. If the developer works on the software every day, an unfamiliarity of 0.0 is used in the calculations. In contrast, if the developer has never seen the software before, an unfamiliarity of 1.0 is used. Any values in-between are estimated and subject to interpretational deviations. In *Table 1* we present the supplementary table provided in the COCOCMO II model definition Manual [USC: Center for Software Engineering, 2000]. The programmer unfamiliarity has a noticeable influence on the resulting cost estimations and, thus, the cost estimation profits from a more accurate UNFM rating.

UNFM Increment	Level of Unfamiliarity
0.0	Completely familiar
0.2	Mostly familiar
0.4	Somewhat familiar
0.6	Considerably familiar
0.8	Mostly familiar
1.0	Completely unfamiliar

Table 1: Rating scale for programmer unfamiliarity (UNFM) as used in COCOMO II *[USC: Center for Software Engineering, 2000]*.

Familiarity, sometimes referred to as "knowing" is one of the two processes that make up the recognition memory. Recollection, referred to as "remembering", is the second component process [Medina, 2008]. Recollection is the retrieval of details with the previously experienced event. In contrast, familiarity is the feeling that the event was previously experienced, without recollection. Familiarity is a fast and automatic process, whereas recollection is a slow and controlled search process [Mandler, 1980]. It is still being debated, weather recollection and familiarity should be considered as separate categories of recognition memory. Alternatively, they would be considered as effects of stronger and weaker memories [Medina, 2008]. The main reason for the ongoing debate is the difficulty in obtaining separate empirical estimates of recollection and familiarity [Curran, et al., 2006]. Irrespective of the debated theories, familiarity

and recollection are intertwined. As familiarity correlates with recollection and familiarity is difficult to measure, we will mainly focus on recollection.

To analyze a programmers' memory, we look at his recollection of code files he wrote. We want to know how much time passed since his last edit of a file and how well he remembers it. Most developers seem to forget the specifics of their source code as time passes. Some argue, that they don't need to remember the concrete code for them to remember the purpose and model architecture behind the file. Nevertheless, the rate of which developers forget the projects information can be described by the forgetting curve hypothesized by Hermann Ebbinghaus in 1885 [Ebbinghaus, 1885]. He extrapolated the hypothesis of the exponential nature of forgetting with Equation (1).

$$R = e^{-\frac{t}{S}} \tag{1}$$

R is the resulting memory retention, S is the relative memory strength, and t is the delay between study and test. The Memory strength is a value that represents how well a subject can remember learned knowledge. The memory retention stands for the percentage of the initially learned knowledge that can still be recalled after the duration of t. This results in the typical forgetting curves we illustrate in Figure 1.

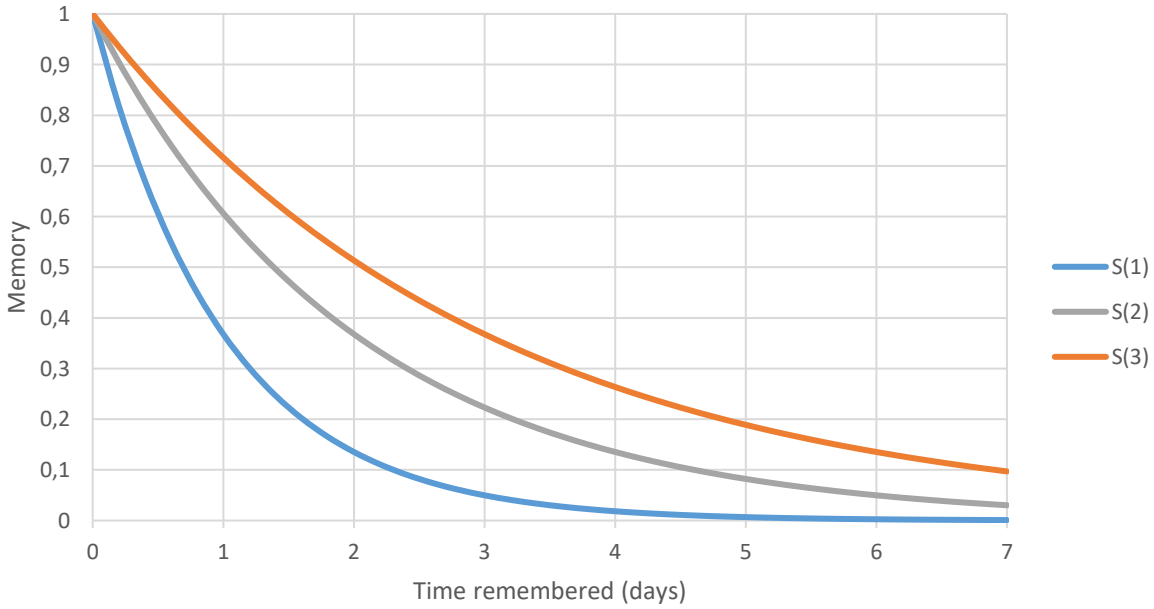

Figure 1: Forgetting curve with different memory strengths s = 1, 2, 3.

The blue curve represents the memory retention over seven days with a memory strength of one. After one day, the memory has decreased from 100% to 37%. The gray curve represents the memory retention with a memory strength of two. It takes the gray curve two days for it to decrease its memory to 37%. The orange curve on the other hand has a memory strength of three and can therefore recall 37% after three

days. It shows how information or knowledge is lost over time if the individual makes no attempts to retain it. How fast the information is lost depends solely on the memory strength, which is unique for every subject and information-type learned.

Although the psychological community has been debating over the exponential nature of the forgetting curve, its possible successors [Lee, et al., 2011] don't make any major difference to the following approximations. The most important difference is an asymptote other than 0, indicating a permanent memory gain. We will use the forgetting curve, introduced by Ebbinghaus [Ebbinghaus, 1885], as it is most widely used. Independent from the used model, the memory strength depends on several factors, such as the meaningfulness and difficulty of the learned material. Some physiological factors like stress and sleep also contribute to the memory strength. To minimize the participant's effort, we will not acquire said physiological factors and generalize the learned material to daily written code.

With the forgetting curve, we can estimate the remaining knowledge a developer has of its file. A factor is still missing, as developers often work multiple times on a single file. Revisiting a file retains his knowledge and alters his forgetting curve, as fainting memories are reinforced [Ebbinghaus, 1885]. The forgetting curve gets shallower each time the individual reviews the information, visualized in Figure 2.

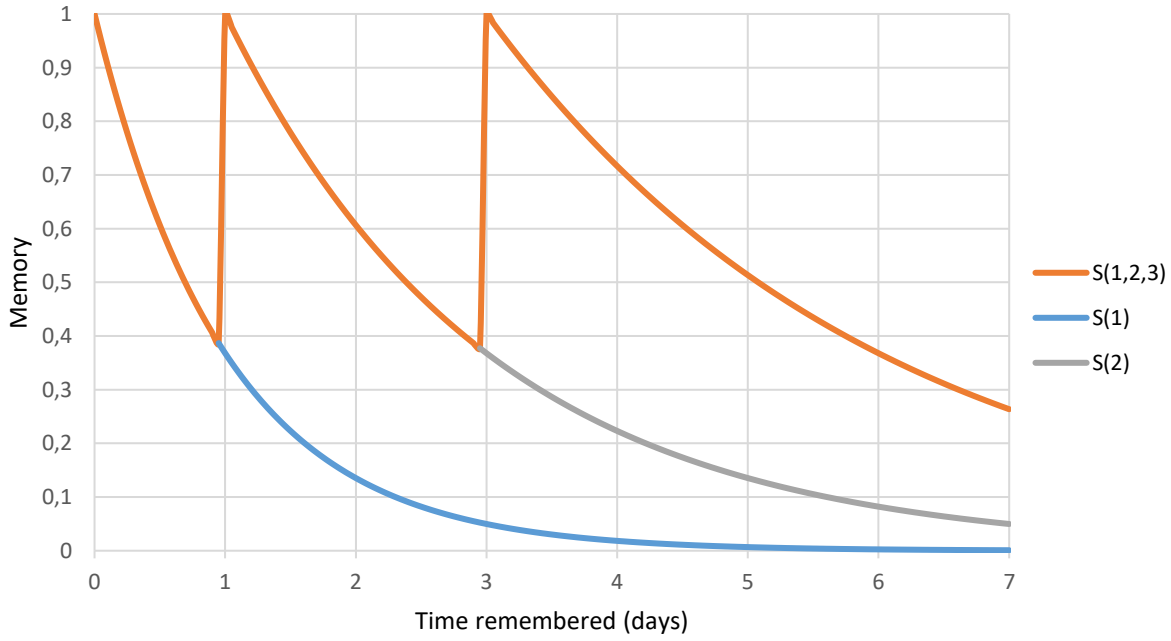

Figure 2: Forgetting curve with two reviewing phases.

Each review reinforces information, prolonging the total time of memory. Nevertheless, it is unknown how well each review works. In Figure 2 we exemplary assume, that every review increases its memory strength by one. The participant starts with a memory strength of one and his memory would decrease following the blue curve. After one day, he only recalls 37% but after the repetition he once again recalls 100%. Because of the repetition, his memory strength increased to two and therefore his forgetting rate follows the shallow gray curve. On the third day, he once again recalls only

37% and actively reviews the given data to recall 100%. The second repetition increased his memory strength to three, shallowing the curve even more. The used values are exemplary but demonstrate the main principle.

Despite the factors influencing the memory strength, it does not vary much between individuals. In contrast, it differs more strongly for different types of knowledge. The forgetting curve is nearly linear for high school classmates' and teachers' names, but street names and random syllables follow the classic Ebbinghaus curve [Bradburn, et al., 2000]. Although there are some minor deviations that come from people training their memory strength or using reinforcing techniques, the average memory strength for writing code should be obtainable as we target only a single type of knowledge.

The forgetting curve has been around since 1885 and forms the foundation of our familiarity algorithm as it approximates the memory at a specific point in time. By taking reviews into account, the results of the algorithm should increase in precision.

3.3 Familiarity Approximation

In this section, we will overview the algorithm. Up to this point, we chose a function that approximates the remaining familiarity of a developer after a period. In the following we will break the project down to analyzable parts, calculate how familiar the developers are with it, and aggregate it back to the entire project.

Software projects consist mainly of code files. These files themselves consist of code which was written and uploaded to the VCS in parts, called commits. In practice commits contain changes of code to multiple files, but we will handle them as separate commits to process files independently from each other. We will additionally ignore code that was removed or overwritten in a newer commit. What we are left with, are commits that contain only additions on a single file level. Each commit contains the source code added, its author and the time it was committed.

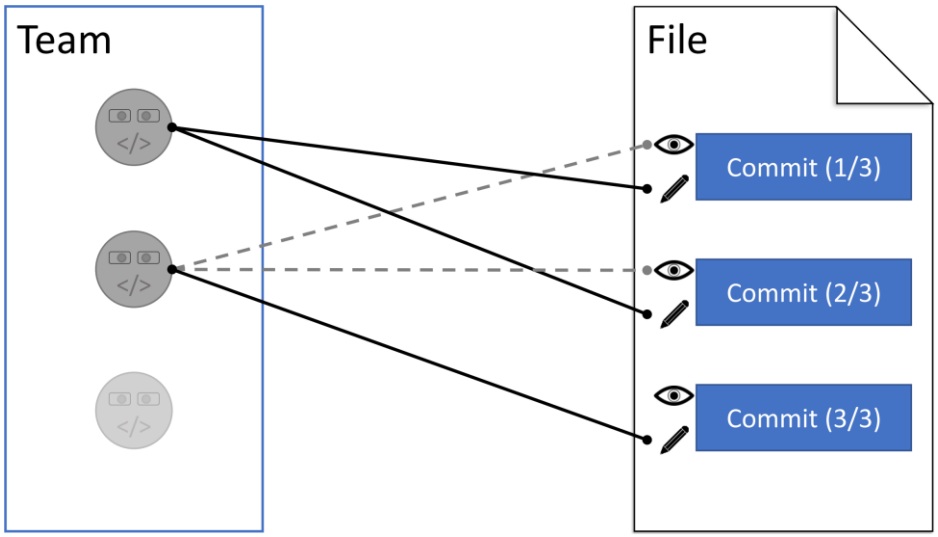

Figure 3: Developer - Commit relationship.

In *Figure 3* we can see the relationship between the developers of a team with a file they wrote. The file contains three commits that were written in rising sequential order. Each commit has an author that is symbolized by the link between a developer and a small pen symbol to the left of the commit. Above each pen symbol is an eye symbol. A dotted line between the developers and the eye symbol means the developer has read the linked commit when he wrote his own commit. Writing a commit implies reading it.

The familiarity between a developer and a commit makes up the basis of our algorithm, as it later aggregated to the entire project. The familiarity is calculated with the forgetting curve by using the time it was committed. After time passes, the written commit is slowly forgotten, until the developer chooses to write another commit in the same file. In that case, he reads his former commit to build upon it and write his second commit. By rereading the first commit, he reviews his knowledge of, leading to an increased familiarity of that commit. In total his first commit's familiarity was increased and he is familiar with his newly written commit.

If a second author decides to add code to the file containing the two previous commits, he should read both commits beforehand. After reading those commits, he gets partially familiar with them. He isn't as familiar as if he wrote them himself, yet his familiarity will rise if he continues to add more code. As time passes, this might lead to the second developer being more familiar with the first two commits than the author himself.

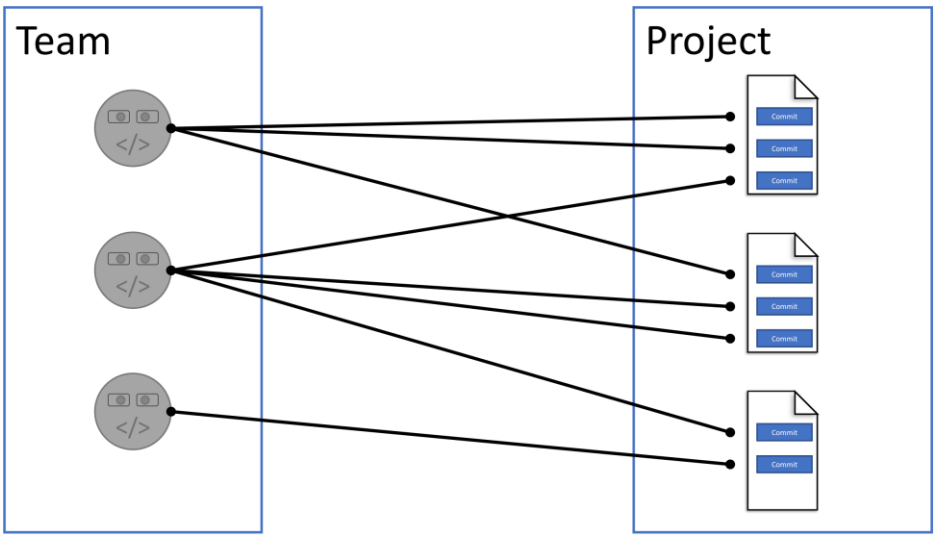

Figure 4: Developer – File relationship.

In Figure 4 we visualize a team that consists of three developers working on a project containing three files. Each file's code is divided into multiple commits, visualized as blue rectangles. Each commit is linked to the developer who is most familiar with it. As the entire project is made up of commits, we have identified the most familiar developer for every line of code.

After calculating the familiarity for each commit, we scale them in relation to the entire project and sum them up. We scale them with the help of their line numbers in comparison to the total project's line numbers. The calculated familiarity tells us how familiar the entire team is with the project.

3.3.1 File Familiarity

As our goal is to analyze how familiar the developers are with the entire project, we need to start by calculating the familiarity for a single file for each author. By aggregating the resulting file familiarities, we get the teams familiarity for their entire project.

$$R_a(c_i, t) = \begin{cases} \min\left(1, \quad e^{-\frac{t-time(c_i)}{S}} + R_a(c_{i+1}, t)\right), & if \quad author(c_i) = a \\ & \quad and \quad c_{i+1} \in C; \\ R_a(c_{i+1}, t), & if \quad author(c_i) \neq a \\ & \quad and \quad c_{i+1} \in C; \\ e^{-\frac{t-time(c_i)}{S}}, & if \quad author(c_i) = a \\ & \quad and \quad c_{i+1} \notin C; \\ 0, & otherwise. \end{cases} \tag{2}$$

$$C = \{c_0, c_1, c_2, \dots, c_n\}$$
$$A = \{a_0, a_1, a_2, \dots, a_m\}$$
$$author(c) = \ author\ of\ c$$
$$time(c) = commit\ time\ of\ c$$
$$\forall c_i, c_{i+1} \in C(time(c_i) \prec time(c_{i+1}))$$
$$i, n, m \ \in \mathbb{N}_0$$

In Equation (2) we calculate the familiarity of an author (a) for a list of commits (C) that make up a file. As we want to know the familiarity of every author, or developer that worked on that file, we need a list of authors (A). For every commit, there is exactly one author, whereas an author of a file has at least one commit. The function *author(c)* returns the author of the commit provided and the function *time(c)* returns the time that commit was submitted to the VCS. The commits are ordered increasingly by submit time, therefore c_0 was written before c_1.

We start the equation with the first commit of the file and the first author and want to know the familiarity for that combination for a given time t. We need to differentiate between four cases. The first case is used if the current commit was written by the developer we are analyzing (author(c_i) = d) and there is a next commit in the list of commits ($c_{i+1} \in C$). That means, that the targeted developer had the opportunity to review his original commit. This case calculates the retention rate with Equation (1) and recursively adds the result of Equation (2), this time using the following commit to look for further reviews. The *min* operation assures that the result is never above one, as the knowledge may increase through reviews, but may never exceed 100%.

The second case is used if there is a following commit but the current one was not written by the developer we want to analyze. In this case, the developer had no opportunity to review his original commit and therefore we look at the next commit.

The third case means that our developer was active again but there are no future commits. Therefore, we calculate the knowledge gain due to this review and end the function. If the file contains only one commit written by the targeted developer, this case would also trigger.

The last case is necessary if this is the last commit and it wasn't written by the targeted author. This case leads to no review phase, and therefore the developer gains no knowledge.

To calculate the file familiarity, we would have to scale the resulting values according to their number of code lines. This is done in the sample below.

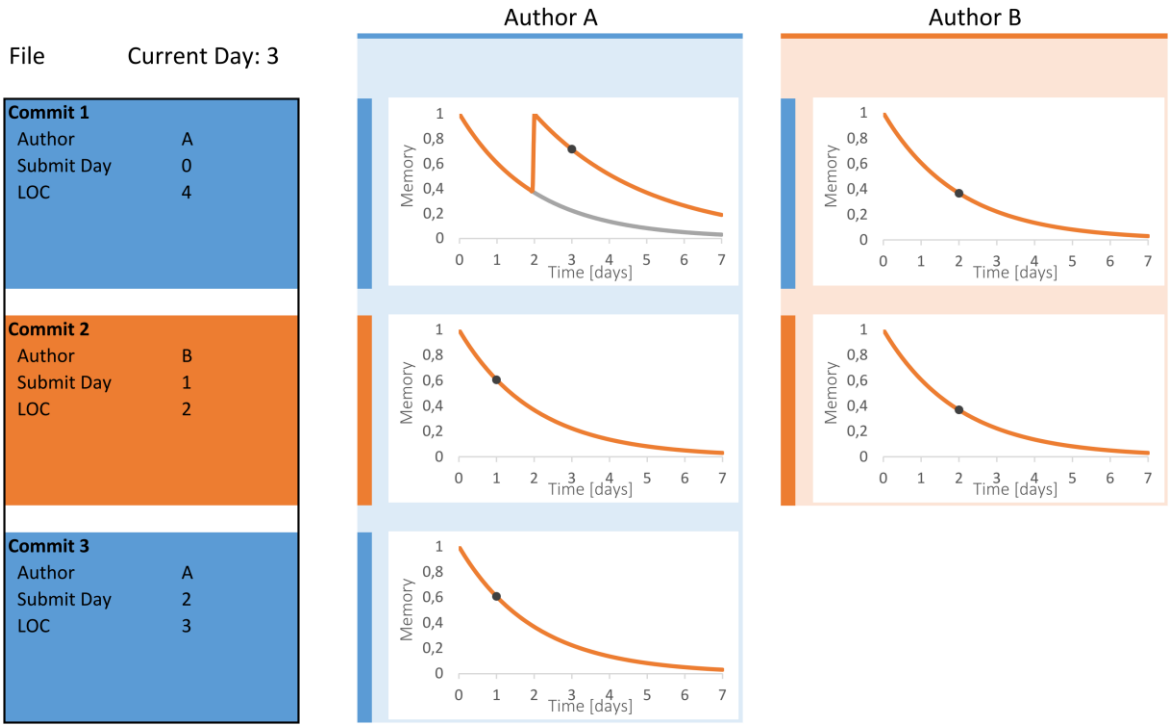

Figure 5: File familiarity example with two authors at day 3.

In Figure 5 we demonstrate an example for a single file with two contributing authors. The file on the left side of the figure contains three commits, each written on a different day. The first commit was written by author A at day 0 which leads to him being familiar with that commit, visualized in the first graph below the caption "Author A". The second commit is written by author B at day 1. As he probably reads the first commit to make his additions, he familiarizes with that commit, shown in the first graph below his caption. The second graph below author B stands for his own commit. When author A writes his third commit, he rereads his first commit, which leads in a spike in the first graph below his caption. He also reads the second commit, written by author B resulting in the second familiarity graph below his caption. The third graph shows his familiarity with commit 3, which only he has. The amount of code lines in each commit is visualized by the number of rows each commit has.

In this example, we chose a memory strength (S) of 2 for demonstration purposes only. In practice, we expect values between 40 and 60.

The algorithm described above results in a list of familiarities for each author and commit. The familiarity for an author and file at a specific time is the sum of each commit-familiarity multiplied by the fraction of code lines in ratio to the total files *lines of code*

(LOC). To calculate the familiarity for the example in *Figure 5* for author A we calculate his familiarity for the first commit with Equation (2). As he is the author of the first commit and there are following commits, we use the first case of the equation. This calculates a remaining knowledge of 0.22 and adds possible reviews by calling itself with the following commit. The following commit was not written by author A and therefore the second case is triggered, calling the equation with the third and last commit. This last commit was once again written by author a and therefore uses the third case, adding knowledge due to this review phase. In total his familiarity for the first commit is about 0.83.

Calculating author A's familiarity for the second commit results in the second case of Equation (2), as there are further commits, but the current one wasn't written by him. Increasing the targeted commit to the third one, reveals no further commits, nevertheless the current one was written by author A. This leads to the third case, as he read the second commit writing the third one. In total this results in a familiarity of 0.61. The third commit leads straight to the third case of the equation, resulting in a familiarity of again 0.61.

After calculating the commit familiarities, we can now scale them to get the file familiarity. The first commit makes up 4/9 of the file's code lines, thus we scale it by 4/9, resulting in a familiarity of 0.37. The second commit makes up 2/9 of the file, resulting in a familiarity of 0.13. The last commit makes up 3/9, resulting in a familiarity of 0.2. Finally, we can add the individual file familiarities up to a file familiarity of 0.7.

In this section, we introduced the equation that enables us to calculate how familiar a developer is with the commits of a file. Out of these values we can now calculate a relative familiarity for the entire file, or as we will see in the following chapter, calculate the familiarity for the entire project.

3.3.2 Project Familiarity

Up to this chapter, we chose a function that describes the memory retention over time and used it to calculate how familiar a developer is with his commits. Building upon the commit-familiarity, we will aggregate it to approximate how familiar the entire team is with its project.

As the project is made up of files, which themselves are made up of commits, we can aggregate the commits directly to the project. By taking the sum of each commit-familiarity from a single developer, scaled with its proportion of LOC in relation to the entire project's LOC, we get the relative project-familiarity for that developer. The resulting number seems relatively small, as it describes how familiar a single developer is with the entire project. To calculate the team's familiarity, we add each developers project-familiarity.

The resulting project-familiarity may exceed 100%, as the developers also familiarize with commits that aren't their own and thus the team's developers may have overlapping commits. In practice this is rare, as files don't have more than a few editors, and most teams divide their work equally.

In the evaluation, the intermediate developer project-familiarity will be compared to the survey results to validify the familiarity approach.

3.4 Summary

In this chapter, we linked the familiarity of a file to the duration that file can be recalled from memory. This process of memory retention can be described by Ebbinghaus' forgetting curve. To apply it to a project, we must break it down to a granularity for which we can automatically retrieve the author, when it was written and how much was written. Fortunately, most projects use a version control system, which fulfills these criteria. Commits, which are the smallest units to contribute code to a project, don't only contain the code, but also its author and time it was committed.

Further, we introduced a formula to calculate the commit-familiarity. This considers reviewing phases, in which the authors read the file's content while, or before adding their own code. Reviewing phases refresh the developer's knowledge of the file, increasing their familiarity.

By aggregating the commit-familiarity and scaling it in proportion to its LOC, we can calculate a file-familiarity and a project-familiarity for a single developer or the entire team.

4. Experimental Design

In the last chapter, we introduced our concept on how to estimate the familiarity of a developer with his code-files. We gather all commits that were submitted to the project's VCS and use the time they were submitted to estimate the amount of knowledge that was forgotten up until the current date. The memory retention rate is then combined with the amount of code the developer submitted to estimate its impact on the total file.

There are a few unknown factors that influence the previously introduced algorithm. For one, we need to approximate an average memory strength for a software developer. This will then be used to calculate the rate the developers forget their code. Additionally, we suspect an increase in said memory strength if the subjects revisit a file. After multiple reviews of a single file, they might remember it far longer than after only a single review, but we do not know if this phenomenon can be applied to software development.

If we can acquire a generally valid memory strength for software developers and include a possible increase due to repeated reviews, we can create a general model that estimates a developer's current state of knowledge towards his projects. In addition, we could estimate his knowledge of a project he once worked on and how it compares to his colleagues.

In this chapter, we will introduce the survey we use to approximate the average memory strength of developers and to search for any increases due to multiple reviews. The determined average memory strength will be used to calculate each participant's project familiarity, which will then be compared to his self-assessment provided through the questionnaire.

To evaluate the survey, we need to combine the results with the actual work the participants submitted to their projects. This provides us with all the details required to estimate the developer's memory strength. To facilitate and partially automate the

data collection, we implemented a tool, which we will explain in detail in the upcoming sections.

In the following sections, we will start by explaining our research questions, followed by the setup of the survey which will help evaluate said questions. After each question's purpose is explained, we introduce the tool that will help evaluate them.

4.1 Research Questions

Determining a developer's knowledge of a file or project is essential for assigning development or reengineering tasks. Managing a whole team requires more effort and a higher degree of knowledge of each developer's capabilities and history. The approach on automatically gathering each developer's degree of knowledge based on their personal development history enables a more detailed insight on each developer's capabilities.

The following research questions will help validate whether the data provided by a VCS is sufficient to approximate a developer's familiarity of their project. In addition, they will give insights on future optimizations.

RQ-1 What is the average memory strength of a developer?

RQ-2 Does the familiarity increase with each edit of a file?

RQ-3 How well can a developer's project familiarity be derived from the files he edited?

Answering the first question (RQ-1) enables us to approximate the familiarity for any developer with equation (1) (Chapter 3.2 – Forgetting Curve, page 13). This step is necessary as the forgetting curve requires a memory strength to calculate the rate of memory retention. As the memory strength does not vary much between individuals, an average is sufficient for further calculations. The second question (RQ-2) is equivalent to answering whether a developer reviews a file while editing it. If the familiarity increases with each edit as stated in the second question (RQ-2), it will dictate the technique used to measure the memory strength for the first question (RQ-1). The result may further be used to increase the precision of the familiarity algorithm. The results of the third question (RQ-3) shows the precision of our algorithm and whether the edit history provided by a VCS is enough to approximate a developer's familiarity. Besides identifying the most valuable developers, it may provide insights into a developer's specializations by looking at the parts edited.

4.2 Survey Setup

The survey covers multiple aspects of our thesis. On the one side, we must retrieve an average memory strength to use it in the forgetting curve. On the other side, we can take the participant's answers to verify the familiarity algorithm approach.

For our participants, we want a broad variety of developers to ensure representative data. We also want different project types in relation to used programming languages to ensure language independency. As our concept relies heavily on version control systems, we will choose the developers of one of the most used, publicly accessible VCS. To achieve a high reply ratio, we will use only projects that are currently in development and are featured for their popularity. Of those developers, we invite only those who worked on the project for the past year. This approach aims at acquiring quantitative data through the amount of developers invited and yet tries to ensure qualitative answers by inviting only active developers. In addition, we favor projects with a scientific background, as research has shown that education is related highly to the response ratio and number of answers participants give to questions [Dey, 1997].

As all our participants have access to the internet, we choose a web service to host the survey. The main disadvantage of internet surveys is that people quit the questionnaire if they feel it is too long [Creative Research Systems, 2016]. To address this issue, we will keep the questionnaire as short as possible.

To invite the project's developers, we will fork the project and automatically parse all commits for their author's email address. As currently most developers use a personal email address rather than using GitHub's proxy email service, we can use these addresses to send the survey's invitation link to. The few cases in which the developers use the proxy email service, the retrieved addresses end in 'users.noreply.github.com' and must be dropped.

To further increase the response rate, we create one survey for every project. This way, we can further personalize the invitation mail and the survey.

In addition to the memory strength and algorithm verification questions, we add two check questions (Questions 5 and 6) to identify respondents that are just "clicking through", which leads to low-quality results. Both answers are estimates of a project's details, which will be compared to the actual project information we will acquire with the tool we introduce in the following section.

4.2.1 Survey Questions

The questionnaire is introduced with a short definition of source code familiarity:

Software familiarity - Generally known as a result of study or experience.

If familiar, you know:
Purpose of a file,
Usage across project,
Structure or programming pattern

The following ten questions are presented one question at a time, each mandatory.

Question 1

"Please enter your GitHub username or email for us to analyze your answers."

The first question asks for the users GitHub username or email address, as the questionnaire service we use does not track from which email address the link was opened. With the help of the username or email address, we can identify the user and therefore his commits. Many users publicly store their name in their GitHub account, which may concern some participants. For this reason, we will ensure the anonymization of their data as soon as we run it through our analysis tool.

Question 2

"How familiar are you with the entire project?"

This question requires the participant to estimate his familiarity for the entire project. The answer must be on a scale from one to ten. To help the decision process, we note that one is equivalent to knowing about ten percent of the project and ten being equivalent to knowing the entire project.

We ask for the project familiarity to verify our algorithm after we have retrieved an average memory strength. The results will be used to answer the third research question (RQ-3).

Question 3

"Name a file from the project that is familiar to you"

With this question, we want the participant to name a file that he is familiar with. We assume he has edited it, which will be checked by our tool. We explicitly ask the participant to not open the chosen file for the following questions. Having the participants GitHub username and a file he worked on, we can look up the file and see all commits he made.

Question 4

"How well do you know the content of the file {answer3}?"

The fourth question requires the user to pick a familiarity level, ranging from one to nine, for the file he chose in question 3. To facilitate the decision, we note that one is equivalent to the participant only remembering the purpose of the file. The answer five is equivalent to knowing the purpose, but also the structure and usage of the file. The answer nine on the scale means he knows the purpose, structure, usage and all the functions of the file.

This question provides us with a file familiarity for the file the participant chose. With this result, we have the author's last commit for this file and the familiarity, which enables us to calculate his memory strength with Equation (1). The results will be used to answer the first research question (RQ-1).

The resulting memory strengths may be misleading, as we can't consider the learning effect that comes from reviewing the file. This is due to the unknown increase in memory strength a review may lead to. To filter these answers and answer the second research question (RQ-2), we will analyze the correlation between the number of edits with the resulting memory strength.

Question 5

"How many lines of code does it contain?"

Question five forces the participant to estimate how many LOC the file he chose contains. This check question will be evaluated by automatically retrieving the correct LOC number and comparing it to the answer. A high error rate may indicate a non-reliable file familiarity estimation as the answers might have been partially wrong.

Question 6

“When was the last date you edited the file {answer3}?”

This question asks for the last date the participant edited the file he chose in question three. The resulting date can again be automatically retrieved and helps to identify unmotivated answers or bad memory. A high error rate will indicate a non-reliable file familiarity estimation.

Question 7

“If this file was lost, how much faster could you rewrite it?”

With this question, we want to see if a high familiarity leads to developers being able to reproduce the file faster. The possible answers range from one to ten and stand for factors of speed. If a two was chosen, that means the author could rewrite the file twice as fast. This explanation was also included in the questions description. We expect hardly any participants to answer with values higher than two. Nevertheless, we don’t want to enable float values as this question is already hard to answer as it is.

Question 8

“After how many days do you only remember the structure and purpose of a file, but have forgotten the details?”

This question tries to retrieve the memory strength from another perspective, as it asks for the number of days that must pass for the participant to have a familiarity equivalent to five on the scale introduced in Chapter 2.3.

As this question is difficult to answer, we expect high deviations. Nevertheless, it should result in a memory strength average between 20 and 80.

Question 9

“If you edit a file of another developer (more than 10 lines), how familiar are you with it afterwards?”

Question nine targets our assumption that you remember your own code better than reading someone else’s code. Also, it could show that developers read the content of a foreign file while editing it. If they don’t the average would be very low.

Question 10

"How well do you track changes other developers make on your files?"

The last question asks how well developers track changes that have been done to files they once wrote. Our assumption is, that only a minority of the developers notice it if another developer changes their file and even less are interested in knowing what they edited. The possible answers ranged from zero, meaning they don't track those changes to ten, tracking every change.

4.3 Analysis Tool

The survey requires us to invite hundreds of developers that have submitted code in the last half year. The email addresses can be acquired through the commits, but accessing these means searching every project. To facilitate this procedure and to reduce the workload on further evaluations, we implemented a tool.

Requirements

The first use case the tool should handle, is the querying of all active developers for a given project. This implies retrieving all commits that were submitted to the project. Out of these commits, we require a list of all developers to later match them to the survey responses. In addition, we need a list of all files that make up the project to find the file the survey response provides us with.

To check the line count, last edit and lines written we need to identify the correct file by the name provided. As the survey responses are likely to contain misspelled or incomplete file- and author names, the tool needs to include a search functionality for partial matches. The file hierarchy should also be visible to differentiate between files with the same name.

As our algorithm calculates the familiarity up to a specific date, we need to assure that the project is in the same state it was on the day the participant answered the survey. This means that our tool needs a date input which will be considered for every familiarity calculation.

Implementation

The tool has been implemented as a Windows Presentation Foundation (WPF) app, which enables us to easily create a windows desktop app running on the .NET framework. This native approach facilitates the access to the file system and third party tools such as the GitHub command line tool "hub". To increase the project analysis performance, we will clone the projects locally instead of using GitHub's web service API.

As we only need the commits that were not overwritten by other commits, we can use the "git blame" command of the GitHub client. This command returns all the infor-

mation GitHub has for a given file on a single line basis for the revision which last modified the line. This includes the author name, author mail, time submitted, commit message, lines of code, filename, and further details. It does not provide any information on lines that have been deleted or replaced.

<table>
<tr><th>Git Blame Output</th><th colspan="3">Usage</th></tr>
<tr><td>Filename</td><td rowspan="2">Total file LOC</td><td colspan="2"></td></tr>
<tr><td>Lines of code</td><td rowspan="3">Lines written</td><td rowspan="3">LOC deviation</td></tr>
<tr><td>Author name</td><td rowspan="2">User identification</td></tr>
<tr><td>Author email</td></tr>
<tr><td>Time submitted</td><td rowspan="2">Commit identification</td><td>Time of code edit</td><td>Last edit deviation</td></tr>
<tr><td>Commit message</td><td colspan="2"></td></tr>
</table>

Table 2: Usage of the queried git blame output.

The usage of the git blame output is illustrated in Table 2. The first step from the git blame output towards usable data models is the aggregation from single line information to entire file commits. To achieve this, we identify a commit by its time submitted and commit message. In addition, we will make sure it came from the same author. To match a commit author with the answered username from the survey, we will primarily use the author email and author name. The author name can be freely chosen on each commit, why we must group them. They will be handy as many users choose a variation of their GitHub username which we can use to identify a survey response.

The filename will be used to identify the file the participant names in the survey. The lines of code will be counted and compared to estimation of the participant on the total file line count. Highly deviating estimations are interpreted as an unreliable answer and excluded from the evaluation (See Chapter 5.2 Results). The time submitted will be used to calculate the last time the participant edited the file. The resulting date is then compared with his estimation on when he last edited the file. High deviations will again be excluded from further evaluations.

To answer the second research question (RQ-2), we will compare the amount of commits a user submitted to the project with his memory strength at the time of his response. This memory strength can be calculated by using the forgetting equation (1). The timespan used is the duration between the survey answer and his last edit, and the familiarity is provided by his survey answer to question 4.

We will approximate an average memory strength, and thus answer the first question (RQ-1), by calculating the memory strength for every participant. This is again done by using the timespan between his last commit and the survey response in combination with his answer to how familiar he is with the chosen file. To reduce the influence

of increased memory strength due to reviews, we consider only participants that edited their file less than five times. This restriction is only for RQ-1, as these results are crucial for RQ-2.

Once we have acquired an average memory strength, we will use it to calculate the participant's familiarity for the entire project and compare it to the answer he gave to question 2: "How familiar are you with the entire project?". His project familiarity will be estimated by calculating and aggregating all file familiarities as described in Chapter 3.3.1. The result will give an assessment on how well the used memory strength works on predicting a developer's familiarity with a file or project and thus answering the third research question (RQ-3).

4.4 Summary

In this chapter, we formulated three research questions that must be answered to validate our familiarity algorithm. The first question searches for a general memory strength to estimate a developer's memory retention of files he worked on. The second question targets the behavior of aforesaid memory strength when the developers review a file multiple times. Finally, the third question aims at whether the algorithm proposed can approximate a project familiarity for a developer by using his project's version control system information.

To answer our research questions, we designed an online survey that targets developers of publicly accessible and version controlled projects. With the help of the version control software, we identify active developers which we invite to participate in the survey. The survey requires the developers to name a file and answer questions regarding their familiarity with the file.

The answers will be evaluated with the help of a tool written for this purpose. It will gather additional information about the participant's work, which will be used to answer our research questions. The additional information will also be used to filter out low-quality answers.

In the following chapter, we will evaluate and discuss the results of the survey and raise possible threats to validity.

5. Evaluation

In this chapter, we describe the evaluation, which we carried out based on the survey and the tool introduced in the previous chapter. We will start by presenting the survey execution including the statistics of the participants and exclusions. Thereupon we present our survey results and use them to answer our research questions. These will each have its own subsection in which the used data is visualized as it was retrieved from the survey and after it was processed to fit our requirements.

To answer whether the memory strength rises with the number of edits to a single file, we will analyze the dependency between the familiarity the participants answered and the number of edits. This will be evaluated by calculating their correlation coefficient.

We will use to different approaches to retrieve the memory strength from our survey results. The first one uses actual files the participants worked on, whereas the second one relies solely on their self-assessments.

Once we have a value for a developer's memory strength, we will use it to calculate each developer's familiarity of the entire project. The results will be compared to the answers the participants gave to the question on how familiar they are with the entire project.

In the following discussion, we will interpret our results and explain the observed deviations. Finally, we discuss possible threats to validity for our approach on acquiring a general familiarity for software developers.

5.1 Survey Execution

In this section, we present the details regarding the execution of the survey conducted in Chapter 4.2 : Survey Setup. The survey was live for two weeks in which the invited participants had their chance on answering it. In the first three days after we sent the invitations, 78% of all results had already been submitted.

The selected projects are listed in Table 3, including the language they are written in and the number of currently active developers. The use of five JavaScript projects reflects the current popularity of the language, making up 38% of GitHub projects rated with over 1,000 stars. Java and Python are also under the top four languages with 12% and 10%. [Codementor, 2016]

Project	Language	Active Developers[3]
aframe	JavaScript	43
angular.js	JavaScript	75
astropy	Python	41
ember.js	JavaScript	75
FeatureIDE	Java	10
ipython	Python	33
odoo	Python	135
react	JavaScript	153
serverless	JavaScript	89
sympy	Python	68

Table 3: Selected projects, the used languages, and currently active developers.

We invited 722 developers by email to answer our survey, consisting of ten questions and taking an average of 3:28 minutes. Of the invited developers, 32% visited the survey, of which 78 completed it. Across all ten projects, on average 10.8% of all invited people answered. As suspected, projects with an academic background had higher answer rates, for instance FeatureIDE (40%) and astropy (31.7%).

The records of 18 participants were removed after applying exclusion criteria to ensure high data quality. The criteria and number of occurrences can be seen in Table 4. As we did not specifically tell the participants to choose a file they worked on, four developers chose a file that they never edited. Nevertheless, they stated they were familiar with it. Although we only invited developers that edited a file in the last year, nine chose a file they had not worked on for over a year.

		Criterion	n
Chosen file	Edited provided file?	No	4
	Time since last Edit	>365	9
Check Questions	Last edit deviation	>100%	5
	LOC deviation	>100%	7
	Total (Criteria not mutually exclusive)		**25**

Table 4: Exclusion criteria and number of occurrences.

[3] State of 12.01.2017

In addition to the excluded answers due to unfitting file selections, we excluded answers that did not meet our quality requirements. These were identified by check questions, which asked for the date the file was last edited by the participant and the approximate number of code lines. Answers that deviated by more than 100% were interpreted as low-quality. In the cases in which we excluded an answer, we dropped the entire questionnaire of the affected participant. As can be seen in Table 4, we had 5 occurrences of edit deviations and 7 of LOC deviations. In total, we excluded 18 survey participants from further evaluations, which left us with 60 valid completed surveys (77%).

5.2 Results

In this section, we will answer each research question in its own subsection. We start by analyzing the dependency between the familiarity and the number of commits to answer whether recurring reviews influence memory strength. By filtering the results, we then search for a general software developer's memory strength. Finally, we test our familiarity approach by calculating the project familiarity for each participant and comparing it to the project familiarity they stated in the survey.

5.2.1 Familiarity and the Number of Commits

To acquire a general memory strength out of our questionnaire results, we had to know whether repeated edits to a single file influence the familiarity perceived by the participants. In case repeated edits influence the memory strength, we would have to exclude all answers with repeated edits in our search for a general memory strength, decreasing our data significantly.

We expected a monotonically rising dependence between the number of edits and the familiarity stated by the participant. We derive the monotonical nature from the experiment conducted by Arthur M. Glenberg [Glenberg, 1976]. He describes a monotonic function for uncued recall at non-short term retention intervals, as we have in our case.

To analyze the correlation, we calculated the Spearman's rank correlation coefficient (Spearman's rho), which is widely used for statistical monotonic dependencies with data that is not normally distributed [Hauke, et al., 2011]. The resulting coefficient lies between +1 and -1 inclusive, where 1 is a positive correlation, 0 means no correlation, and -1 is a negative correlation.

The calculated correlation for times edited and stated familiarity is $r(58) = 0.67$, which indicates a strong positive monotonic correlation. To validate our result, we conducted a t-Test to test whether the calculated Spearman's rho is significantly more likely than the null hypothesis. In this case, our null hypothesis is, that there is no correlation between the number of edits and the stated familiarity. The test result was positive with a significance value (p-value) of less than 0.001, indicating that there is a significant positive relation between the number of edits and the stated familiarity. Therefore we

dismiss the stated null hypothesis and assume that there is a relation between the number of edits and the stated familiarity.

Additionally, we plotted the graph, displaying the number of edits and the associated familiarity, which can be seen in Figure 6. The great amount of high familiarities despite a low number of edits indicate that an increase in familiarity may occur with the increase in edits, but there are other reasons besides the number of edits.

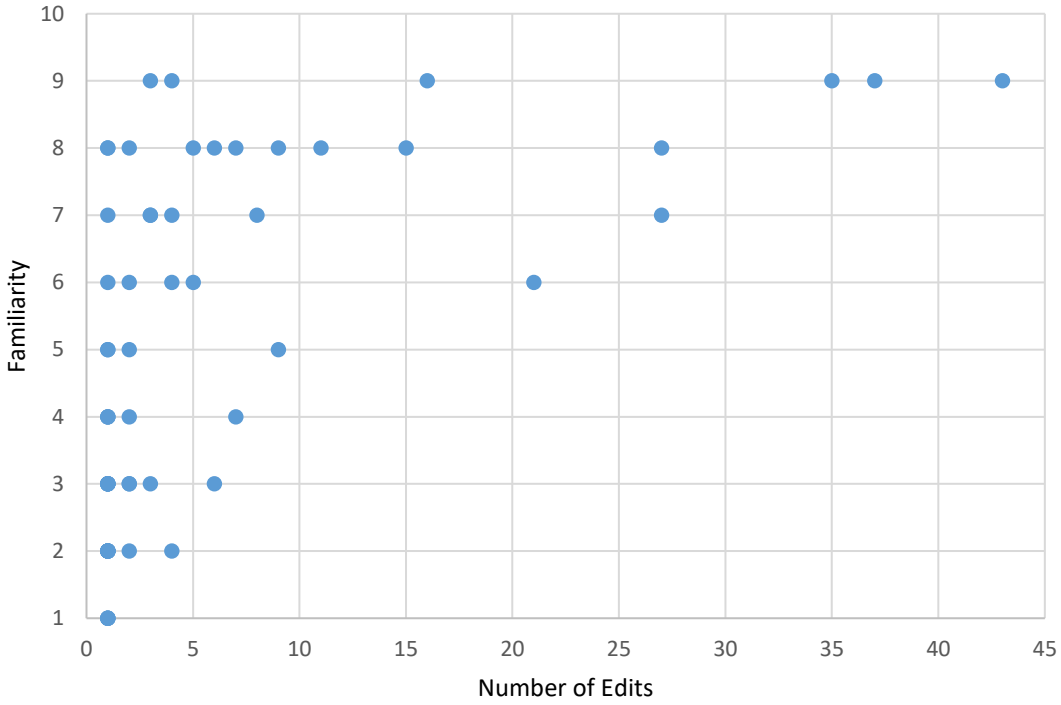

Figure 6: Number of edits in relation to the stated familiarity.

The increase in familiarity that comes with the increase in edits comes from the gain of knowledge that comes with each review of the file. Each time the developer rereads his file to prepare himself to write new code, he refreshes his knowledge and strengthens his memory of the knowledge he still remembers. This behavior was shown by Ebbinghaus in his simplicities experiments [Ebbinghaus, 1885]. Yet, there are many developers that state a high familiarity for a file they have only edited once or twice. This is no contradiction to our statement. Possible reasons for this are explained in the upcoming discussion section.

5.2.2 General Memory Strength for Software Developers

As we now know that the familiarity is dependent on the number of edits, we need to exclude familiarities in which the participant edited the file for more than a few times to get a general memory strength for a single edit.

Firstly, we calculated each participant's memory strength using the forgetting curve formula from Chapter 3.2 and transposing the equation (1). The resulting equation can be seen in Equation (3).

$$S = -\frac{t}{\ln(R)} \tag{3}$$

For the duration (t) we used the timespan between the last edit of the participants and the submit date of the survey in days. For the familiarity (R) we used the familiarity the participants stated. As the questionnaire asked for an integer value between 1 and 9 (including 1 and 9), but the familiarity the formula returns is a percentage, we had to scale the results. We deliberately scaled the results to match 10% - 90%, as a value of 0 cannot be achieved as all participants were asked to choose a file they were familiar with. A familiarity value of 0 would mean they are not at all familiar with it, missing the meaning of the question. On the other hand, a value of 10 would mean they edited the file the same moment they answered the question.

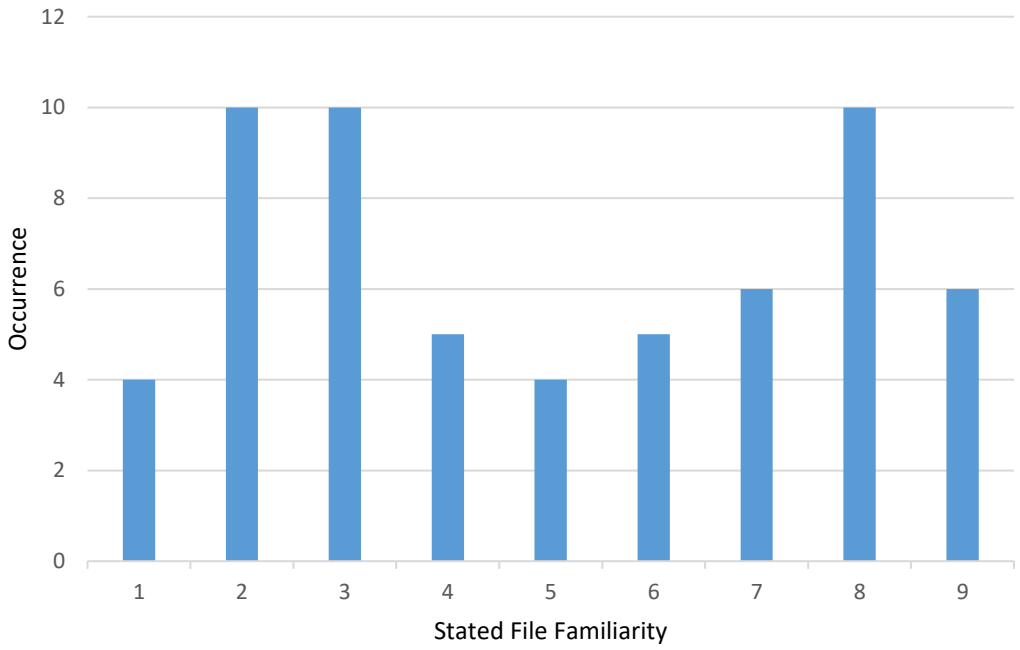

Figure 7: File familiarities stated in the survey.

In Figure 7 we visualized all familiarities that were stated in the survey. The peaks in the distribution at the lower part (2-3) and the higher part (8) indicates a binary polarity amongst the participants. They seem to be either relatively unfamiliar or almost fully familiar.

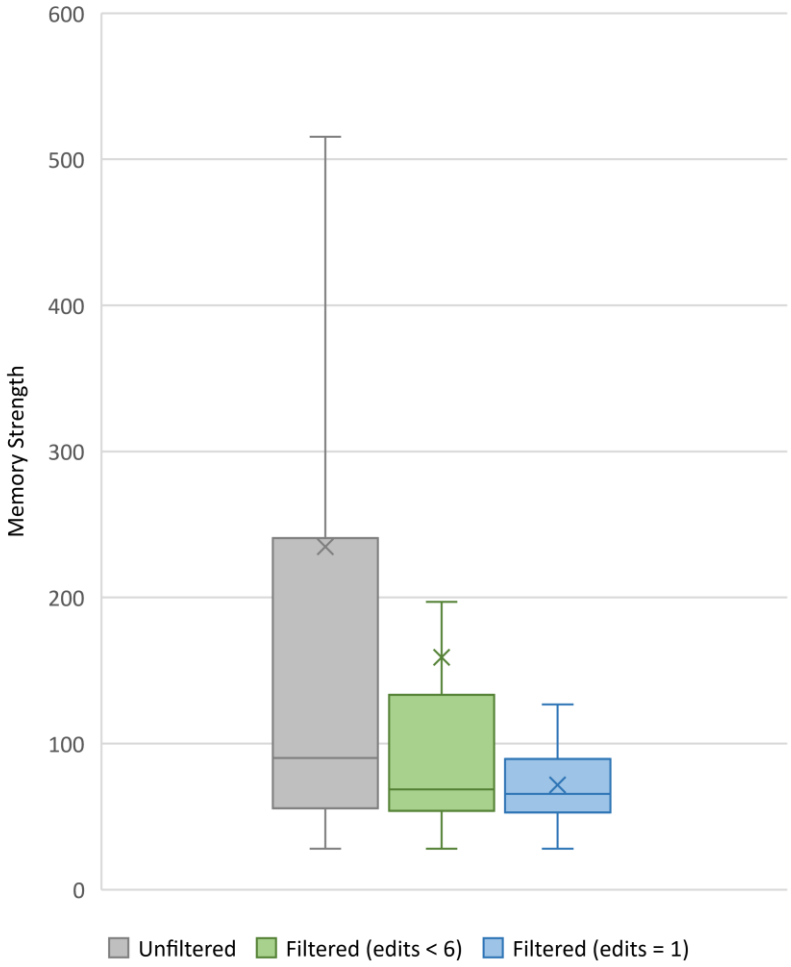

Figure 8: Boxplots of the unfiltered and filtered memory strengths.

In Figure 8 we displayed the resulting memory strengths as box plots. We can see, that the median (horizontal line in each box) seems to get closer to 65.5 as we filter our results by the number of edits. The mean (marked by a cross in the diagrams) starts far greater than the median, but as we filter further, the outliers get sorted out, resulting in an approximation towards the median. As we are searching for a value that represents the memory retention after a single edit, we will take the median value resulting from the filtered plot on the right side of Figure 8, which is 65.5. To put this value into context, this means that after 65.5 days, a software developer remembers only 36.7% of his originally remembered file. After approximately 45 days, half of his knowledge is forgotten.

To support our estimation, we implemented a second approach by letting the participants indirectly estimate their own memory strength. We asked the survey participants to state after how many days they only remember the equivalent of half of their file (Question 8). This estimation is hard to answer, as the relatively big deviation of the answers shows. The results can be seen in the boxplot in Figure 9.

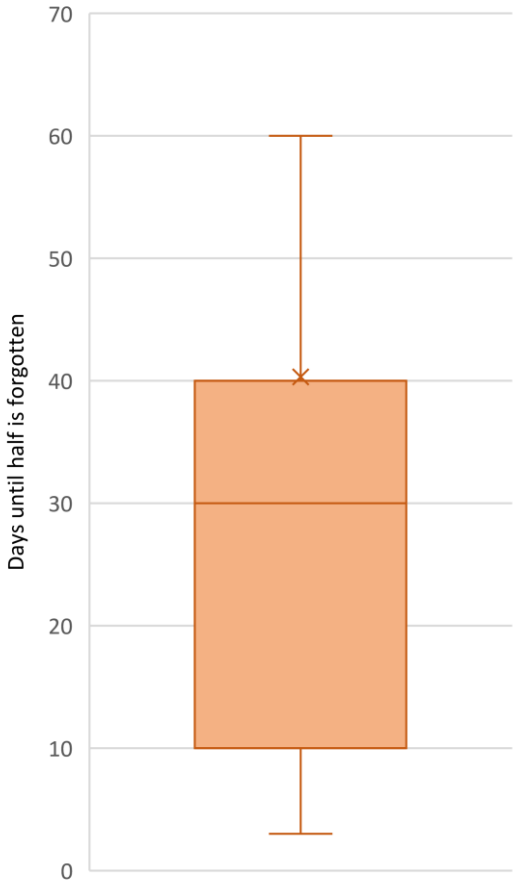

Figure 9: Days until half of the knowledge is forgotten.

The mean is symbolized by the cross above the box, which is about 40.3 and the median is visualized by the horizontal line inside the box, which is 30. Most of the results seem to be guessed in weeks (i.e.: 7, 10, 14, 20, 30, 45, 60, 90, …). The exact answers can be seen in the appendix.

To calculate the equivalent memory strength, we used equation (3) and set the familiarity to R = 0.5. For the duration (t) we used the median as the distribution is widely spread, resulting in a memory strength of about 43.3.

To better compare the results of both approaches, we converted the stated durations to memory strengths, as done before. The result can be seen in Figure 10, in which we compared the self-assessment to the previously retrieved memory strength. We can see, that both value areas are different to some extent, with their median at 43 and 65, which narrows down a generally applicable memory strength, with some deviation.

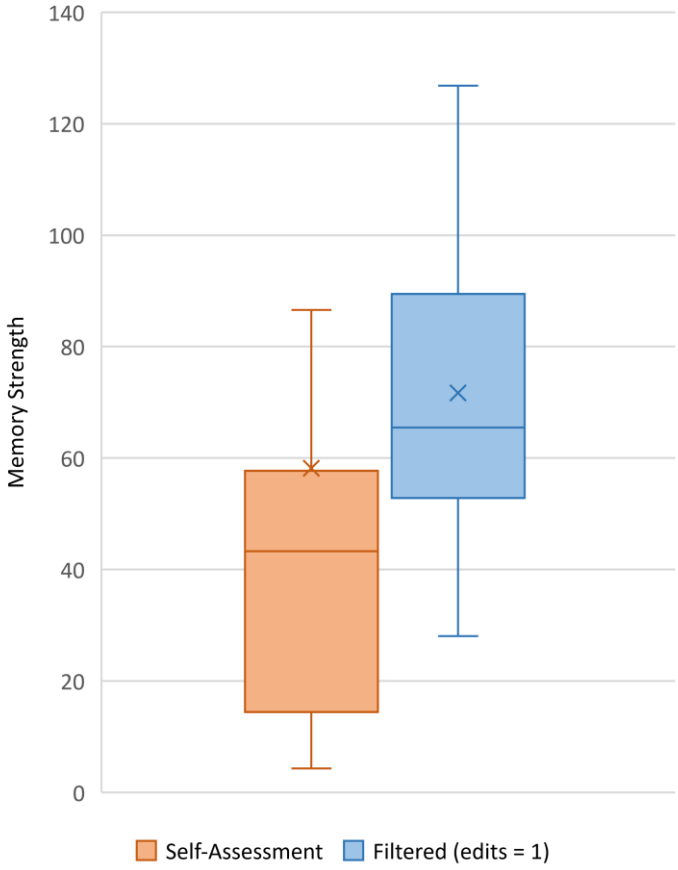

Figure 10: Filtered memory strengths besides the self-assessed memory strengths.

For the following calculations, we will be using a memory strength of 65, as the first calculation is not only based on self-assessment, but also on the familiarity rating of a file the participants worked on and remembered. In addition, we have some concerns over some of the participants' answers in the pure self-assessment, which we will explain in the discussion section.

5.2.3 Project Familiarity

To answer our third research question, we need to measure how well a developer's familiarity for an entire project can be estimated by analyzing the files he edited. In our case, we analyzed the developer's edit behavior and derived from it his familiarity for the files he worked on. By aggregating all his file-familiarities, we approximate a familiarity for the entire project, as described in Section 3.3.2.

In the survey conducted, the participants answered how familiar they are with the entire project (Question 2). In contrast to the previous familiarities, this question asked for a percentage of the project they are familiar with. The possible values ranged from 1 (10%) to 10 (100%). The results are visualized in Figure 11. The distribution shows that most of the participants are not familiar with much of the project, rather they are only familiar in their region of expertise.

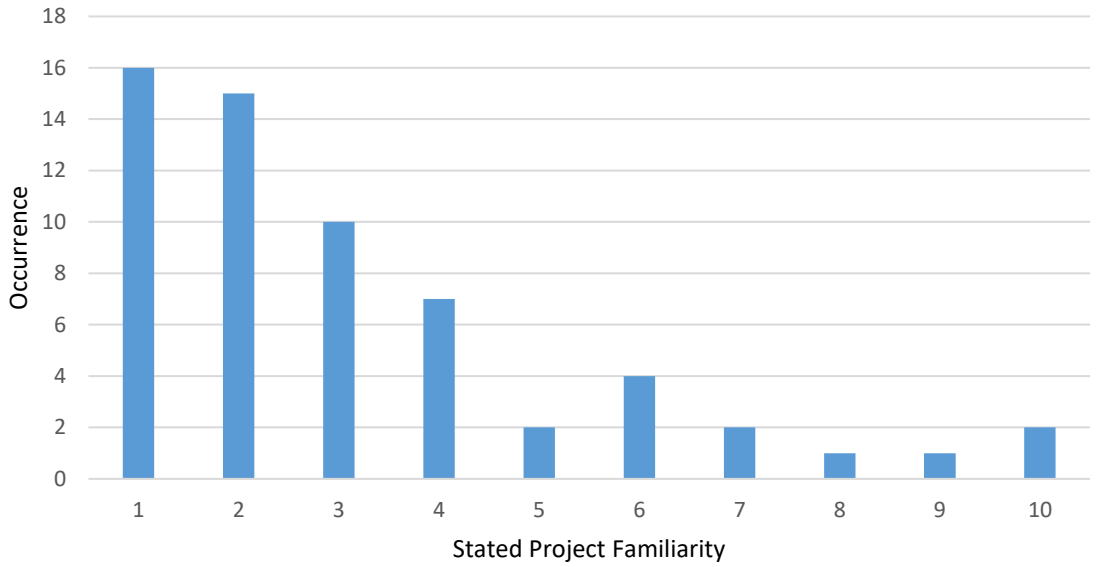

Figure 11: Project familiarities stated in the survey.

To test how well our algorithm calculates the project familiarity, we used it to calculate each participant's project familiarity and compared it to the stated project familiarity. The similarity was once again calculated with Spearman's rank correlation coefficient, as we have only limited information over the nature of the data distribution. The resulting correlation coefficient was $r(58) = 0.74$, indicating a strong correlation between the two variables. Our null hypothesis is that the datasets are not correlating, which is why we calculated the significance level of our correlation. The resulting significance value (p-value) of $p < 0.001$ is far smaller than our threshold of 0.5, which means that our null hypothesis is unlikely to be true. In contrast, there is a significant positive relationship between the calculated project familiarity and the one stated by our survey participants. Therefore we dismiss the stated null hypothesis and assume that there is a relation between the calculated project familiarity and the stated project familiarity.

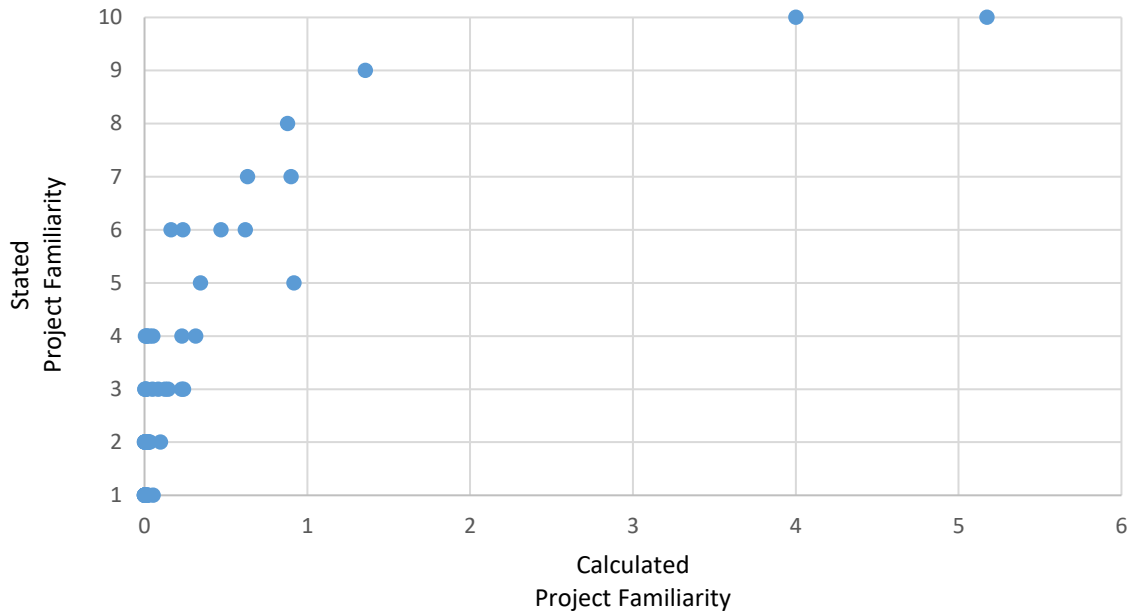

Figure 12: Calculated Project Familiarity compared to Stated Project Familiarity.

Figure 12 shows how the calculated project familiarities (X-Axis) are compared to the stated ones (Y-Axis). Although there is a strong positive correlation between them, this graph is deceiving as most of the elements have a stated familiarity of one to three. The dots in the graph are overlapping, hiding their number. More interesting are the values above four. To the look of the plot, there seems to be a linear increase up to nine. After that we have only two participants who stated to know the entire project. As our algorithm only considers the code they have worked on, the calculated values of 4 and 5 indicate that these two developers were involved in almost half of the entire project's code. Having contributed this amount of code, the developers are perhaps more familiar with the project than our algorithm can detect. This might indicate an additional source of familiarity that is not dependent on the committed code.

5.3 Discussion

In this section, we will discuss the results of our survey we presented in the previous section. Therefore, we will review our research questions from Chapter 4.1 and answer them with the information retrieved from the survey. In addition, we will discuss any deviations from the expected results.

RQ-1 What is the average memory strength of a developer?

To answer the first research question, we asked the survey participants for a file they are familiar with and to state the degree of familiarity for that file. By calculating the

timespan between his last edit and the day he answered the questionnaire, we had all the information to calculate the associated memory strength with Equation (3). The resulting median memory strength was about 65. To further support any results, we implemented a second question in the survey that askes the participant indirectly for his self-assessed memory strength. The results were not filtered this time, leading to a higher significance. On the other hand, the answers suggest that the question was not precise enough or too hard. The results varied from developers stating that they forget half of the file after only 3 days, up to 360 days. Nevertheless, the median of the calculated memory strengths is about 43.

The distribution of the stated familiarity values, visualized in Figure 7, may indicate some difficulty on the self-assessment. It seems, as if the participants generalized from the desired value range of 1 to 9 to an easier assessment of "not that familiar" (2-3) or "pretty familiar" (8). Nevertheless, there are enough values in between that suggest proper self-assessments.

Both approaches resulted in different values. We prioritize the result from the first approach, as the values do not deviate as much as they do in the second approach. We interpret this as a better understanding of the question and thus a more reliable result. The resulting memory strength of 65 is equivalent to forgetting half of a files content after about 45 days, which seems plausible to interviewed software developers.

RQ-2 Does the familiarity increase with each edit of a file?

To answer the second research question, we had to prove a positive relation between the familiarity and the number of edits. We achieved this, by retrieving the number of edits from the file the participant named in the survey and comparing it to the stated familiarity of said file. With a t-test we showed a significant positive relation between the familiarity and the number of edits. The participants that state a high familiarity for a file they have only edited once don't contradict our statement. An increase in familiarity may also occur if the file was recently edited.

Our results show that the memory gain that occurs when reviewing information can be transferred to the process of repeated file edits. Each time a developer edits a file, he refreshes his knowledge of the file and thus strengthens his knowledge of it.

RQ-3 How well can a developer's project familiarity be derived from the files he edited?

The third research question builds upon the results of the former questions by using the acquired memory strength and calculating how familiar a developer is with his

project. The result is then compared to the answer they provided in the survey on how familiar they are with their project. By proving a positive correlation between both values, we showed that our algorithm can be used to calculate a developer's project familiarity. In addition, by using only the data retrieved from the version control system, we showed that the information contained is enough to derive a developer's project familiarity.

The values we calculated are very small in comparison to the numbers the participants stated. This may be partially due to a general over estimation of the participants, but also due to our algorithm not considering project knowledge that exceeds the plain code writing. For instance, our algorithm cannot perceive if a developer is supervising or managing the development of other developers and thus is familiar with code that goes beyond his own code files.

5.4 Threats to Validity

In this section, we will elaborate on the internal and external threats to validity.

Internal Validity

Although we had a high number of participants, we had to exclude some of the answered questionnaires due to misunderstood questions. We should have conducted a pilot study to avoid any misunderstandings. Nevertheless, the resulting number of 60 valid results is enough to be representative.

To answer the first research question, we had to further filter our results. We were lucky to obtain 29 values, which was enough to answer our question. Nevertheless, we could have obtained more results by formulating our questions more precisely. This would have overall improved the quality of our results.

During the evaluation, we identified multiple users that submit code using multiple GitHub accounts. In the identified cases, the users contributed using one account and after some time, they continued using a new one, not ever contributing with the old one again. Unfortunately, we could not identify these users automatically, which may have led to some incomplete project familiarities. We identified a total of two cases in which the email address was equal enough to suggest the same developer. Therefore, we only see this as a minimal threat.

External Validity

The external validity targets any threats that concern the abstraction from our study participants to the general developer community. In our case, we made sure to invite developers from some of the biggest projects from the open source community. Nevertheless, this may also be regarded as a possible threat to validity, as these GitHub

projects are mainly developed by volunteering distributed developers. Due to this distributed nature of open source development, we should regard our participants as standalone developers rather than traditional commercial software developers.

Some participants contacted us after the survey, to note that they do not regard all the project contributors as full developers. They argued that there are plenty of contributors that have only added a single line. We can partially agree with their statement, as we also observed some participants that named a readme file in which they edited less than four lines. In our opinion, these edits are not enough to qualify the authors as average developers. Fortunately, we encountered only four of those participants. For future studies, this should be an exclusion criterion. Nevertheless, we included these answers as our algorithm does not differentiate between the amount of code that has been contributed.

The programming languages that the used projects are written in are JavaScript, Python and Java. They are amongst the most used languages which makes it representative for most developers. The size of the projects range from small projects like FeatureIDE with 39 contributors to react, which was created by Facebook with 957 contributors.

We may have marginally biased our results, as we chose two projects (FeatureIDE and astropy) for their academic background and not for their popularity on GitHub. They make up 18% of our results.

5.5 Summary

In this chapter, we presented the results of our survey including the statistics of the execution. After elaborating on the exclusion of some of the questionnaires, we used the data to answer our research questions.

We started by showing a strong significant relation between the number of edits and the stated familiarity. This indicates that the source code development behaves equally to reviewing learned facts, which Ebbinghaus investigated in his experiment, resulting in his forgetting curve [Ebbinghaus, 1885]. In addition, we encountered familiarity for files that have been edited only once, which indicate that the familiarity cannot be calculated solely by using the number of edits. Nevertheless, we showed that the familiarity rises with each edit of a file, answering our second research question.

Thereupon, we estimated an average memory strength for software developers by using two separate approaches that use different answers from the survey. The results were inconclusive, as the values vary from one another. They suggest a value in the area from 43 to 65. We answered our first research question by using a memory strength of 65, as we have more reasons to trust the results acquired through one of the approaches.

With the acquired average memory strength, we calculated the project familiarity for each participant and compared it to the project familiarity they stated. Our test resulted in a significant positive correlation between both project familiarities, indicating that our algorithm can be used to approximate the project familiarity. By answering our third research question, we realized that our approach is missing a source of familiarity that is not contained within the files the developers edited. Nevertheless, the edited files provide the main source of familiarity.

Finally, we discussed deviations and conspicuous distributions among the survey data. The threats to validity which we identified were discussed in the last section. We identified misunderstood questions and a few non-representative developers as the main threats to validity.

6. Related Work

In this chapter, we describe the related work that has been done prior to ours. We start by stating the main approach that most of the current work is based on. In each section, we introduce a different approach and the benefits it brings with it. Finally, we explain which parts of the approaches our algorithm uses and how it differs from all of them.

Most automated approaches on determining the familiarity or expertise of developers rely solely on change information. They build upon the "Line 10 Rule", which is a heuristic that attributes the developers who changed the file most often the highest expertise. One example is the Expertise Browser [Mockus, et al., 2002] which gathers and ranks developers' expertise over time. They aggregate their results to make statements concerning modules or even find experts across multiple projects. An alternative tool, the Expertise Recommender [McDonald, et al., 2000] uses the same heuristic and ranks the experts according to the commit time, so that the last developer that modified a file is the most familiar.

These approaches only regard the files the developers edited, but not the files they had to understand in order to use them. The Emergent Expertise Locator [Minto, et al., 2007] refines the approach of the Expertise Browser by considering files that were often changed together. This way, it widens the area of expertise by also considering relevant files that the developer did not change himself. In addition, Minto et al. introduced monotonically increasing familiarity over time as the developers commit changes.

Girba et al. [Gîrba, et al., 2005] consider finer-grained information by equating expertise with the number of code lines that each developer changes. This enables them to make more precise statements about a single file.

To approach the lacking information about the usage of files, Schuler et al. [Schuler, et al., 2008] analyzed the usage of API methods. This method enabled them to identify

the developers that have most expertise in the usage of certain API methods. They differentiated between implementation expertise and usage expertise, which enabled them to recommend experts for files with no or little history.

To further consider the usage of files, Fritz et al. [Fritz, et al., 2010] introduced a Degree-of-Knowledge model that differentiates between the Degree-of-Authorship and Degree-of-Interest. The Degree-of-Interest they introduced cannot be calculates solely with version control system information, but requires an extension that monitors the developer's actions. Their method enabled them to consider every time a developer opens a file and whether they make changes to it, even if they do not commit them to the project.

The previously mentioned approaches get better the more information they can analyze. The approach we consider in this work relies only on the version control system's information, which is why we only detect submitted changes to the file. Considering files that are often checked in together would provide some additional information, but it is hard to filter out irrelevant relations which distort the results. We went further than making assumptions on a file basis, as we calculate the familiarity for every committed change. This way, we acquire an increased granularity that helps to identify knowledge of developers that had a small impact on the file.

Our approach targets the nature of memory retention, that all the previous papers disregard. As projects grow in size and duration, we can no longer just compute the expert based on how much he edited, but must consider how much he has already forgotten. Our concept could be added to the existing expert mining algorithms, to simulate the memory retention.

7. Conclusion

Determining the familiarity or expertise of software developers can be used to identify experts in a project. In addition, the degree of familiarity for a project is also used to estimate the efforts of a project. Software product lines can be used to reduce the costs of development, support and customization of multiple variants. To achieve these benefits, some additional work must be invested [Pohl, et al., 2005]. Thus, companies should calculate how much the additional investment influences future product costs. To achieve this, there are multiple ways of estimating the costs, including the cost-model-based approach. This approach relies on manual calculations, including the estimation on how familiar the team is with its project.

In this thesis, we introduced a method to automatically calculate a developer's project familiarity by using the project's version control system. Our concept differs from the previous expert mining approaches, as it regards the memory retention that occurs over time. In the case of transitioning towards a software product line, there are old projects that need to be refactored or reengineered. Conventional approaches don't regard the forgetfulness of the former developers, which may lead to higher costs when the developers need to refamiliarize with the old project. In some cases, it could be more effective to start from scratch if there are no developers left that are familiar with the project.

7.1 Contributions

In Chapter 3 we derived our requirements from which a cost-estimation-model would most likely benefit from. Based on the requirements, we decided that the forgetting curve from Ebbinghaus [Ebbinghaus, 1885] would be adequate to describe the memory retention over time. In combination with the data that a version control system provides, we introduced an equation that calculates the familiarity for file based on the author's

edits. In addition, we presented a method to aggregate the file familiarities based on the amount of code submitted to acquire a familiarity value for the entire project.

Within Chapter 4 we formulated three research questions that had to be answered to evaluate our algorithm. To answer these questions, we relied on the data of currently developed projects. We required both the project details and the self-assessment of the developers. To acquire this information, we planned a survey that asks a developer for information concerning a single file of his choosing. We implemented several different approaches to acquire our information, strengthening our results. In addition, we had to implement a tool to facilitate further evaluations.

In Chapter 5 we evaluated our survey. We started by reviewing the survey execution statistics and naming the exclusion criteria that improved our overall survey quality. We identified a positive correlation between the familiarity and the number of edits, which shows that the memory gain that occurs when reviewing information can be transferred to the process of repeated file edits. Each time a developer edits a file, he refreshes his knowledge of the file and thus strengthens his knowledge of it. Thereupon, we calculated an average memory strength for a software developer. It's value of 65 represents the time required for a developer to forget his code. Consequently, a developer's familiarity decreases to about 36% after 65 days. After we had retrieved the memory strength, we used it to calculate the project familiarity for all survey participants. The results were compared to the results they stated in the survey. A significant positive correlation between the stated and the calculated project familiarity shows that our concept can be used to calculate a developer's project familiarity.

Finally, we discussed the threats to validity which we could identify. These consisted mainly of misunderstood questions in the survey, as the number of participants compensated for the few excluded questionnaires.

7.2 Future Work

During our work, we identified several topics for further research. To start with, we were not able to use all the answers provided by our participants. We planned on evaluating, how much developers review foreign files in contrast to their own. The associated question is question 9. Unfortunately, we had several responses that indicated a misunderstanding of the question, which lead to us dropping it.

The original goal of calculating a familiarity for the entire team could not be proven, as we only questioned individual developers. A future case study of a closed development team could prove whether our approach is suited to estimate a team's familiarity.

During our research, we found some promising expert mining approaches that even consider the files read by the developers. We suspect that these approaches would benefit from our memory retention algorithm.

A. Appendix

A.1 Survey Results

Questions 2-4:

Id	Project	[Q2] Project Familiarity [1-10]	[Q3] Filename	[Q4] File Familiarity [1-9]
1	aframe	2	material.test.js	2
2	aframe	4	camera.js	2
3	aframe	2	scene.md	4
4	aframe	3	embedded.test.js	6
5	angular.js	1	ngOptions.js	2
6	angular.js	3	angular.js	5
7	angular.js	3	forms	3
8	angular.js	9	rootScope.js	8
9	angular.js	4	ngBindSpec.js	2
10	angular.js	2	compile.js	4
11	angular.js	3	compile.js	8
12	astropy	2	lupton_rgb.py	8
13	astropy	7	quantity.py	9
14	astropy	1	card.py	1
15	astropy	1	test_c_reader.py	4
16	astropy	5	core.py	6
17	astropy	2	solar_system.py	1
18	astropy	6	table.py	3
19	ember.js	6	component.js	5
20	ember.js	2	curly-component.js	2
21	ember.js	1	iterable.js	1

22	FeatureIDE	4	ConfigurationBuilder.java	3
23	FeatureIDE	2	FeatureUtils	8
24	FeatureIDE	1	ImagesComposer	3
25	FeatureIDE	4	FeatureModelEditorContributor.java	2
26	ipython	3	Demo.py	8
27	ipython	5	interactiveshell.py	7
28	ipython	8	interactiveshell.py	8
29	odoo	3	odoo-bin	8
30	odoo	4	sale.py	9
31	odoo	1	product.py	1
32	odoo	3	account.py	8
33	odoo	1	danidee10.md	4
34	odoo	10	pos_order.py	5
35	odoo	1	account_tax_data.py	6
36	odoo	2	manifest.py.template	3
37	odoo	2	models.py	9
38	odoo	2	account_asset.py	7
39	react	1	conferences.md	3
40	react	2	documentation	5
41	react	2	addons-animation.md	7
42	react	1	ReactNativeBaseComponent.js	2
43	serverless	6	create.test.js	8
44	serverless	2	updateStack.js	6
45	serverless	7	PluginManager.js	7
46	serverless	2	Variables.js	3
47	serverless	6	awsProvider.js	7
48	serverless	1	intro.md	4
49	serverless	10	PluginManager.js	7
50	serverless	4	Handler.cs	9
51	serverless	1	awsProvider.test.js	2
52	serverless	3	install.js	8
53	serverless	1	docs/README.md	3
54	sympy	4	fancysets.py	2
55	sympy	2	ccode.py	3
56	sympy	1	miscellaneous.py	2
57	sympy	3	matrices.py	6
58	sympy	3	fp_groups.py	9
59	sympy	1	test_numbers.py	3
60	sympy	1	guess.py	9

Questions 5-10:

Id	[Q5] Estimated Lines of Code	[Q6] Estimated Last Edit	[Q7] Rewrite Factor	[Q8] Familiarity = 5 after days	[Q9] Foreign Familiarity [1-9]	[Q10] Change Tracking
1	200	01.03.2016	1	20	7	8
2	170	01.08.2016	5	360	9	10
3	120	25.11.2016	1	40	7	4
4	60	20.12.2016	1	50	5	0
5	700	18.08.2016	1	30	2	0
6	1200	22.12.2016	1	45	6	2
7	500	14.12.2016	1	5	8	9
8	2000	15.12.2016	1	10	6	9
9	300	01.04.2016	3	200	7	3
10	2000	14.11.2016	8	40	8	8
11	4000	15.11.2016	1	60	7	8
12	200	15.12.2016	2	30	6	4
13	1000	01.11.2016	3	10	3	8
14	500	13.06.2016	1	30	4	0
15	2000	09.12.2016	3	14	4	3
16	400	15.12.2016	2	30	5	9
17	800	01.03.2016	1	10	4	4
18	666	09.10.2016	1	30	4	0
19	500	10.01.2017	2	40	5	0
20	300	20.08.2016	1	10	6	7
21	150	19.08.2016	2	5	4	7
22	600	05.11.2016	6	20	6	6
23	1000	01.06.2016	2	90	6	3
24	150	01.08.2016	2	7	5	6
25	150	01.10.2016	1	31	7	7
26	400	10.10.2016	8	30	5	9
27	2000	15.10.2016	1	15	4	7
28	600	24.01.2017	3	30	6	5
29	5	06.01.2017	2	7	6	0
30	800	29.02.2016	2	180	5	6
31	600	01.08.2016	1	30	5	9
32	1300	10.01.2017	6	60	2	5
33	10	15.11.2016	1	10	5	8
34	1500	01.01.2017	1	30	5	4
35	150	20.12.2016	2	5	6	3
36	50	10.11.2016	1	5	7	10
37	6000	13.01.2017	2	90	6	8
38	500	15.08.2016	2	30	5	8
39	100	25.10.2016	3	45	8	5

40	200	11.12.2016	1	30	4	0
41	400	04.01.2017	2	14	5	0
42	150	01.07.2016	1	10	6	3
43	400	02.01.2017	3	90	7	0
44	200	20.12.2016	1	20	5	7
45	200	09.10.2016	1	14	6	7
46	200	20.10.2016	2	5	4	6
47	350	10.12.2016	1	40	5	2
48	100	25.11.2016	1	7	5	0
49	300	13.08.2016	1	3	6	10
50	50	15.12.2016	2	45	4	0
51	500	20.10.2016	1	10	3	8
52	150	10.12.2016	2	15	7	8
53	15	15.11.2016	1	30	7	5
54	1000	15.10.2016	1	10	6	5
55	800	01.06.2016	2	14	4	2
56	700	14.09.2016	1	3	2	2
57	3000	01.01.2017	1	5	4	6
58	2000	01.09.2016	2	40	5	4
59	1000	07.08.2016	3	30	3	8
60	250	01.10.2016	1	200	5	10

Bibliography

Bergey, John, et al. 1996. A Reengineering Process Framework. 1996.

Bird, Christian, et al. 2009. The Promises and Perils of Mining Git. *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on. IEEE.* 2009, pp. 1-10.

Boehm, Barry, et al. 2004. A Software Product Line Life Cycle Cost Estimation Model. *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on. IEEE.* 2004, pp. 156-164.

Boehm, Barry, et al. 1995. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of software engineering.* 1, 1995, Vol. 1, pp. 57-94.

Bradburn, Norman M., et al. 2000. Temporal Representation and Event Dating. *The Science of Self-report: Implications for Research and Practice.* 2000, pp. 49-61.

Chen, Yih-Farn Robin, Rosenblum, David Samuel and Vo, Kiem-Phong. 1997. *System and Method for Selecting Test Units to be Re-Run in Software Regression Testing. 5,673,387* USA, 1997.

Clements, Paul and Northrop, Linda. 2007. *Software Product Lines : Practices and Patterns.* s.l. : Addison Wesley, 2007. 978-0201703320.

Clements, Paul. 2002. Being Proactive Pays Off. *IEEE Software.* 2002, p. 28.

Codementor. 2016. *codementor.io.* [Online] February 19, 2016. [Cited: 3 1, 2017.] https://www.codementor.io/learn-programming/beginner-programming-language-job-salary-community.

Creative Research Systems. 2016. *surveysystem.com.* [Online] 2016. [Cited: February 28, 2017.] http://www.surveysystem.com/sdesign.htm.

Curran, Tim, et al. 2006. Combined Pharmacological and Electrophysiological. *Journal of Neuroscience.* 2006, Vol. 7, 26, pp. 1979-1985.

Dalgarno, Mark and Beuche, Danilo. 2007. Variant Management. *3rd British Computer Society Configuration Management Specialist Group Conference.* 2007.

Dey, Eric L. 1997. Working with Low Survey Response Rates: The Efficacy of Weighting Adjustments. *Research in Higher Education.* 38, 1997, 2.

Dubinsky, Yael, et al. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on. IEEE.* 2013, pp. 25-34.

Ebbinghaus, Hermann. 1885. *Über das Gedächtnis.* s.l. : Books on Demand, 1885.

Fischer, Stefan, et al. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on. IEEE.* 2014, pp. 391-400.

Fritz, Thomas, et al. 2010. A Degree-of-Knowledge Model to Capture Source Code Familiarity. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering.* 1, 2010, pp. 385-394.

Gimnich, Rainer and Winter, Andreas. 2005. Workflows der Software-Migration. *Softwaretechnik-Trends.* 2005, Vol. 25, 2, pp. 22-24.

Gîrba, Tudor, et al. 2005. How Developers Drive Software Evolution. *Principles of Software Evolution, Eighth International Workshop on. IEEE.* 2005, pp. 113-122.

Git. 2017. Git Documentation. *git-blame v.2.12.0.* [Online] 02 24, 2017. [Cited: 03 10, 2017.] https://git-scm.com/docs/git-blame.

Glenberg, Arthur M. 1976. Monotonic and Nonmonotonic Lag Effects in Paired-Associate and Recognition Memory Paradigms. *Journal of Verbal Learning and Verbal Behavior.* 15, 1976, Vol. 1, pp. 1-16.

Goodman, Paul S. and Garber, Steven. 1988. Absenteeism and Accidents in a Dangerous Environment: Empirical Analysis of Underground Coal Mines. *Journal of Applied Psychology.* 1988, Vol. 73, 1, p. 81.

Harrison, David A., et al. 2003. Time Matters in Team Performance: Effects of Member Familiarity, Entrainment, and Task Discontinuity on Speed and Quality. *Personnel Psychology.* 2003, Vol. 3, 56, pp. 633-669.

Hauke, Jan and Kossowski, Tomasz. 2011. Comparison of Values of Person's and Spearman's Correlation Coefficients on the Same Sets of Data. *Quaestiones geographicae.* 30, 2011, Vol. 2, pp. 87-93.

Highsmith, James A. 2002. *Agile Software Development Ecosystems.* s.l. : Addison-Wesley Professional, 2002.

Jorgensen, Magne, Boehm, Barry and Rifkin, Stan. 2009. Software Development Effort Estimation: Formal Models or Expert Judgment? *IEEE software.* 2009, Vol. 26, 2, pp. 14-19.

Kanki, Barbara G. and Foushee, H. Clayton. 1989. Communication as Group Process Mediator of Aircrew Performance. *Aviation, Space, and Environmental Medicine.* 1989.

Kastner, Christian, Apel, Sven and Batory, Don. 2007. A Case Study Implementing Features Using AspectJ. *Software Product Line Conference, 2007.* 2007, pp. 223-232.

Katz, Ralph. 1982. The Effects of Group Longevity on Project Communication and Performance. *Administrative science quarterly.* 1982, pp. 81-104.

Krueger, Charles W. 2002. Easing the Transition to Software Mass Customization. *International Workshop on Software Product-Family Engineering.* 2002, pp. 282-293.

Krüger, Jacob. 2016. *A Cost Estimation Model for the Extractive Software-Product-Line Approach.* 2016.

Lee Rodgers, Joseph and Nicewander, W. Alan. 1988. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician.* 42, 1988, Vol. 1, pp. 59-66.

Lee, Averell and Heathcote, Andrew. 2011. The Form of the Forgetting Curve and the Fate of Memories. *Journal of Mathematical Psychology.* 2011.

Littlepage, Glenn, Robinson, William and Reddington, Kelly. 1997. Effects of Task Experience and Group Experience on Group Performance, Member Ability, and Recognition of Expertise. *Organizational behavior and human decision processes.* 1997, Vol. 69, 2, pp. 133-147.

Luhmann, Niklas. 2000. Familiarity, Confidence, Trust: Problems and Alternatives. *Trust: Making and breaking cooperative relations.* 2000, Vol. 6, pp. 94-107.

Mandler, George. 1980. Recognizing: The Judgement of Previous Occurence. *Psychological review.* 1980, Vol. 3, 87, pp. 252-271.

Martinez, Jabier, Ziadi, Tewfik and Bissyandé, Tegawendé F. 2015. Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach. *Proceedings of the 19th International Conference on Software Product Line.* 2015, pp. 101-110.

McDonald, David W. and Ackerman, Mark S. 2000. Expertise Recommender: A Flexible Recommendation System and Architecture. *Proceedings of the 2000 ACM conference on Computer supported cooperative work.* 2000, pp. 231-240.

Medina, John J. 2008. The Biology of Recognition Memory. *Psychiatric Times.* 25.7, 2008, pp. 13-15.

Menychtas, Andreas, et al. 2013. ARTIST Methodology and Framework: A Novel Approach for the Migration of Legacy Software on the Cloud. *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on. IEEE.* 2013, pp. 424-431.

Mercurial. 2009. Mercurial: The Definitive Guide. [Online] 05 07, 2009. [Cited: 03 10, 2017.] http://hgbook.red-bean.com/read/.

Minto, Shawn and Murphy, Gail C. 2007. Recommending Emergent Teams. *Mining Software Repositories.* 2007, pp. 5-12.

Mockus, Audris and Herbsleb, James D. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. *Proceedings of the 24th international conference on software engineering.* 2002, pp. 503-512.

Northrop, Linda M. 2002. SEI's Software Product Line Tenets. *IEEE software.* 2002, Vol. 19, 4, pp. 32-40.

Pohl, Klaus, Böckle, Günter and Van Der Linden, Frank J. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* s.l. : Springer Science & Business Media, 2005.

Rodríguez-Bustos, Christian and Aponte, Jairo. 2012. How Distributed Version Control Systems Impact Open Source Software Projects. *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE.* 2012, pp. 36-39.

Rubin, Julia, et al. 2012. Managing Forked Product Variants. *Proceedings of the 16th International Software Product Line Conference.* 1, 2012, pp. 156-160.

Saake, Gunter, et al. 2013. *Feature-Oriented Software Product Lines.* Berlin : Springer, 2013.

Schuler, David and Zimmermann, Thomas. 2008. Mining Usage Expertise From Version Archives. *Proceedings of the 2008 international working conference on Mining software repositories.* 2008, pp. 121-124.

USC: Center for Software Engineering. 2000. COCOMO II Model Definition Manual. 2.1, 2000.

Winter, Andreas. 2004. Software-Reengineering — Werkzeuge und Prozesse. *Workshop der GI-Fachgruppe „Software-Wartung".* 2004, Vol. 15.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 30.03.2017