Informatics

Ein Reengineering-basierter Ansatz für die Untersuchung der Benutzbarkeit einer Software-API

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Stefan Hanreich, BSc.
Matrikelnummer 01227486

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab, BSc.

Wien, 29. Juli 2023

Stefan Hanreich　　　　Jürgen Cito

Informatics

Evaluation of a reengineering-based approach for evaluating software API usability

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Stefan Hanreich, BSc.
Registration Number 01227486

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab, BSc.

Vienna, 29th July, 2023

Stefan Hanreich Jürgen Cito

Erklärung zur Verfassung der Arbeit

Stefan Hanreich, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Juli 2023

Stefan Hanreich

Acknowledgements

This work would not have been possible without the help of many people, I want to take this opportunity to thank them:

My parents, Karin and Christian, who have supported me in countless ways during my studies and life.

My sisters, Maria and Carola, who cheered me up when things were not going well and always believed in me.

My late great-uncle Herbert, the only engineer in my close family, who always lent me an ear when I wanted to talk. This work is dedicated to him and his memory.

My uncle Günther who paved my path in becoming a programmer by gifting me my first programming book.

My whole family - my grandparents, uncles and aunts, cousins and many other varying degrees of kinship - who are always a source of counsel and joy.

My friends and acquaintances, listing all of them by name is a futile effort, who provided me with much needed stress-relief during the time I was writing this thesis.

My oldest and closest friends - Martin, Patrick, Lukas, Constantin, Andy, Daniel, Nicol, Simon - every day I am glad to be blessed with your friendship.

My friends Simon and Thomas, who helped me find motivation to go through with my diploma thesis.

My friend Bernd, who was the best study buddy I could have hoped for.

My professor Jürgen and my advisor Markus, who provided invaluable input and guidance.

My colleagues from work - Claudio, Wolfgang, Robert, Sadik, Mira, Stefan and many others - who helped me grow as a software engineer.

The people from the Elektra Initiative who always were eager to help and support me during my endeavour.

Everyone I possibly forgot, I hope there aren't too many - I owe everyone I forgot a drink of their choice.

Kurzfassung

Diese Arbeit untersucht die Praxistauglichkeit einer neuen Methode um die Benutzbarkeit von Software APIs zu untersuchen - einen Reengineering-basierten Ansatz. Bei einer Untersuchung mit diesem Ansatz wird die Software-API von dem Reviewer mit Hilfe von Methoden des Reengineerings neu entwickelt um die Benutzbarkeit der API zu untersuchen und zu verbessern. Diese Arbeit versucht eine Antwort auf die Frage zu finden ob dieser neue Ansatz eine brauchbare Alternative zu den existierenden, konventionellen Methoden zur Untersuchung von Software APIs ist (analytische und empirische Methoden). Um diese Frage zu beantworten, habe ich in meiner Arbeit eine Case Study durchgeführt, die diese 3 Methoden auf die API von Elektra anwendet. Die Case Study hat gezeigt, dass der Reengineering-basierte Ansatz eine Alternative zu den herkömmlichen Untersuchungsmethoden darstellt. Sie hat Probleme mit der Benutzbarkeit der API gefunden, die die anderen beiden Methoden nicht aufdecken konnten.

Abstract

This work explores the viability of a novel software API review method, a reengineering-based approach. When reviewing an API with this method, the reviewer reengineers an existing API to evaluate and improve its usability. This work aims to answer the question whether the newly proposed reengineering-based approach is a viable alternative to existing, conventional API usability review methods (analytic and empirical methods). In order to answer this question I conducted a case study during which I performed three separate reviews on the API of Elektra's core library using three different review methods. The case study showed that the reengineering-based approach can provide an in-depth review of the usability of a software API. It uncovered usability issues that were not uncovered by the other two conventional review methods. The reengineering-based approach proved to be a viable approach for evaluating the usability of software APIs when compared to the existing methods.

Contents

List of Figures

CHAPTER 1

Introduction

The use of third-party software libraries has become ubiquitous in modern software development due to the advantages regarding development speed and cost [MAN+13]. Programmers interact with software libraries through the API they expose. Good usability of those APIs is therefore an elementary goal for designing an API [PFM13], so programmers can easily integrate the library into their own projects. Researchers have proposed several approaches for reviewing the usability of software APIs [RTP19]. Those conventional methods usually focus on performing a review of the software API and subsequently reengineering the API to fix the problems those reviews uncovered [MMD16]. What if we used the reengineering process itself as a method for reviewing the software API, instead of performing the reengineering afterwards? Since reengineering has to happen during the process of improving the usability of a software API anyway, we could utilize this process as a review method. Reengineering is a widely studied field of software engineering and there exist many approaches for reengineering existing software to improve its quality [MQO18]. Our newly proposed review method uses the act of reengineering itself as a review method. This way we can draw upon the processes and strategies that have been developed for reengineering software and use those processes and strategies to transform our existing software API into a more usable one.

CHAPTER 2

Background

2.1 Software APIs & Usability

A software API provides an interface for programmers to access functionality that is provided by a software library. Through the rise of open-source software, the amount of available libraries that can be used by programmers has increased tremendously [DR08]. Because of this, modern software development increasingly relies on the usage of software libraries in the software development process [Ebe09] [SG12]. The usage of software libraries written by a third party is part of many programmers daily life. Using external software libraries speeds up development significantly, while increasing the quality of the software, among many other benefits [MF07]. This development leads to an increasing interest in the question of what makes a software API easy to use for developers.

An important factor for choosing between existing software libraries is the usability of the library [XAT+20]. The usability of a software library affects how easy it is for other software developers to use the provided functionality in their projects. Good usability of a software library speeds up development time significantly. In specific use cases, for instance cryptography, good usability also has important security implications [GS16]. Badly designed APIs of cryptography libraries can lead to severe security issues [GWL+19]. In this case, the usability of a library is even more important because it minimizes the risk of misuse.

Once a software API has been published and is relied upon by its users, it becomes harder to make changes to the API. Changes lead to additional work for users, who have to change their existing software, this in turn leads to greater dissatisfaction. It also increases the effort for the maintainers, especially when they have to make breaking changes to the API. [HRA+15] Those factors make it important to try to eliminate usability issues with the API before widespread adoption. The later an issue appears during the development lifecycle, the harder it is to address it.

There are several definitions on the usability of a software API. Usability describes how easy it is to learn and use the software. This is a simple, less formal definition. For more elaborate descriptions on what constitutes usability of a software one can look at ISO standards 25010 [ISO11]. The standard describes quality models for systems and software. Their quality in use model describes several properties for usable systems: Effectiveness, Efficiency, Satisfaction, Freedom from Risk and Context coverage. The goals can also vary depending on the use case of the software library. Some projects may weigh certain aspects higher than other software projects due to the context they are usually utilized in.

There have been efforts made to create metrics that measure the usability of software APIs automatically. In their work, Rama and Kak [RK15] formulate 9 structural issues affecting the usability of the APIs. They formulated quantifiable metrics for measuring the usability of Java APIs in an automated fashion. Other works try to measure the time that a developer takes to accomplish a certain task using a software library. Endrikat et al. [EHRS14] compared the impact of static typing on API usability by measuring the time it takes test candidates to perform certain tasks. The idea behind this is, with libraries that have good usability developers should spend less time accomplishing certain tasks - compared to less usable software libraries. This makes time spent completing a task an interesting metric for measuring the usability of an API.

There are other approaches that are measuring usability qualitatively rather than empirically and do not rely on the measurement of any metrics. There are certain commonly agreed upon best practices that result from the experience of software developers. APIs that are usable should conform with those, while less usable ones more often violate those best practices. Zibran et al. [ZER11] have compiled a list of usability issues found in bug trackers, that compile the most recurring issues in software projects. These usability factors are usually not measurable or hard to measure in an experiment, such as the metrics proposed by Rama and Kak [RK15]. One instance for this would be Rama and Kak's metric S8, which addresses exception classes that are too general. While it might be possible to automatically flag all general exceptions being thrown, it is hard to automatically check whether the thrown exception is specific enough. There is also a lot of knowledge and nuance involved in picking the correct exception class for IOExceptions in Java for example.

2.2 Reengineering & Redesign

Rosenberg et al. [RH96] describe software reengineering as the process of examining, analyzing and alterating an existing software system to recreate it in a new form. The main objective of re-engineering a software system, according to Sneed [Sne91], is reducing maintenance cost, which in turn is achieved by increasing the quality of the software and reducing the complexity of a software system.

Byrne and Gustafson [BG92] describe a basic process model for re-engineering software that consists of 5 phases: Analysis and Planning, Renovation, Target System Testing, Re-documentation and Acceptance & System Transition, which has also been referenced by Rosenberg et al [RH96].

Based on the process model of Byrne and Gustafson, multiple approaches for reengineering a software system were created. Rosenberg et al. [RH96] as well as Majthoub et al. [MQO18] describe three basic approaches for re-engineering a software system based on the process model by Byrne and Gustafson:

The Big Bang Approach, where an old system gets replaced by a new system completely at one point in time, hence the name 'Big Bang'. The incremental approach tries to rebuild one part of the system after another. The components that make up the old system are reengineered one after another. The evolutionary approach is similar to the incremental approach and tries to replace parts of the old system with a reengineered component. The key difference is that this does not necessarily happen within the structure of the old system, but rather the components for reengineering are chosen based on existing functionality in the old system.

Building upon the foundation for software reengineering processes, many frameworks for the reengineering process have been created. Tahvildari et al. [TKM03] introduced a 'quality-driven software re-engineering' approach that describes a framework for using hard and soft quality constraints to guide the software reengineering process.

2.3 API usability evaluation methods

There are different ways of classifying techniques for reviewing software APIs. For my work I will be sticking to the classification of Rauf et al. [RTP19], who conducted a large meta-study on software API usability review techniques. Rauf et al. performed a literature review of 47 primary studies, that performed API usability reviews. They also tried to classify the guidelines for API usability used in the evaluated studies. Rauf et al. used 2 different methods for classifying usability studies, analytic and empirical, in their systematic mapping review.

There are other ways of classifying usability evaluation methods compared to the ones that Rauf et al. use. One of the widely used classifications has been created by Hartson et al. [HAW01], who define 4 main categories of usability evaluation methods: Expert Evaluation / User Evaluation / Model Evaluation / Evaluation Location. The work of Hartson et al. looks at usability evaluation methods mainly from the perspective of evaluating the usability of interactive designs such as user interfaces - while this work focuses on the usability of software APIs. This is why I will be using the more specific classification used by Rauf et al, who classified the review methods as analytics-based or empirics-based reviews.

There is an example given by Scriven et al. [Scr72], that is also quoted by Hartson et al.:

> If you want to evaluate a tool, say an axe, you might study the design of the bit, the weight distribution, the steel alloy used, the grade of hickory in the

> handle, etc., or you might just study the kind and speed of the cuts it makes in the hands of a good axeman.

This example can be used for describing the two different categories proposed by Rauf et Al [RTP19]: analytic and empirical. While analytic methods with this analogy would focus more on the design and attributes of the axe, empirical methods would be the study of how a good axeman uses the axe in this analogy. Those categories are not mutually exclusive, since some studies combine analytic and empirical methods to measure the usability of an API.

2.3.1 Analytic

Analytic methods usually use the specification of the API for their evaluation of the usability [RTP19]. An example for an analytic methods would be collecting quantitative metrics, which can then be compared against pre-defined target values. An example for such a review is the study conducted by Rama and Kak [RK15], which I already described earlier. Other forms of analytic methods include reviews by experts, who study the API specification and provide feedback on how to improve the usability of the API, such as the study by Farooq et al. [FZ10]. In this study software developers performed moderated peer-reviews of a software API using a cognitive dimensions framework.

2.3.2 Empirical

Empirical methods focus on studying the usage of a software API. Instead of studying the API specification and documentation, the subject of study is how the API gets used by developers. General examples would be task-based tests that measure how long developers take to accomplish a certain task using an API or surveys including users of an API or mining repositories of projects and analyzing the data.

Code Reviews or Code Inspections are a staple of modern software engineering best practices and there exist many studies on how to conduct code reviews / inspections [How06] [SSC$^+$18], as well as their effectiveness [SSC$^+$18] [MKAH16] [BBZJ14].

An example for a study conducted with an empirical method is the study conducted by Nanz et al. [NTPM13]. They compared with a case study how two different groups learn concurrent programming when using different programming languages.

Another example is a study conducted by Murphy-Hill et al. [MHSH$^+$18] which used the change history of source code to analyze the usability of an API.

2.3.3 Reengineering

For this study I propose a novel method for evaluating the usability of a software API: a method based on reengineering the existing API. This method works by reengineering the existing API in a different programming language and then using the lessons learned from reengineering the API for improving the usability of the API.

This is an approach that reverses the usual approach that the empirics-based and analytics-based review use. With those two conventional approaches a review is performed and then, based on those reviews, the API is reengineered to solve the problems that the reviews uncovered. This approach performs the reengineering step first and then uses the insights gained from the reengineering process to identify issues with the current state of the software API. The results from this reenginering process can then be used for improving the usability of the existing API.

Reengineering software is a topic that has been studied for a long time and there exist many models and frameworks for reengineering existing software [MQO18]. I want to apply methods and processes from reengineering methods in this case study. There are several works that focus on rengineering software with a specific emphasis on improving the quality (which includes usability) of software:

Tahvildari et al. [TKM03] propose a 'Quality-driven software re-engineering' method that focuses on optimizing specific quality constraints. Constraints that Tahvildari et al. mention that are particularly important for the usability of software APIs are 'High Code Naming and Commenting Quality' as well as 'High Structure Quality'.

Another work by Moura [Mou17] focuses on 'Awareness Driven Software Reengineering'. Moura defines non-functional requirements that are based on awareness about the environment the software is used in. Particularly interesting for my case study is Social Awareness, where Moura defines goals for the reengineering process that take the needs of the users of the software into account.

CHAPTER 3
Case Study

3.1 Elektra

Elektra describes itself in the introduction [Ini23a] in the following way:

> Elektra serves as a universal and secure framework to access configuration settings in a global, hierarchical key database.

An analogy used in the documentation for Elektra [Ini23a] is that Elektra can be thought of as a virtual file system for configuration files. Configuration values are stored in a global key database (KDB). One can also mount existing configuration files into the KDB. If values for a mounted configuration file get updated in the KDB, then those changes are instantly reflected in the respective configuration files in the file system.

Elektra includes many plugins for accessing configuration files in different formats via its KDB. There are many plugins for mounting configuration files with different formats. Those plugins include support for many formats such as JSON, TOML, INI, many UNIX configuration files (for instance the hosts file) and more. Elektra includes plugins that can be used as validators or filters for values, plugins can fulfill a multitude of purposes and are not restricted to parsing configuration files alone.

Elektra provides a library written in C that can be used to programmatically manipulate Keys as well as access or modify the KDB. There are bindings for many programming languages that make the functionality of the C library available in many languages such as Python, Rust and many more. The libraries are split in different parts: There is a high level API, an API for notifications, the core API, libraries for developing plugins, libraries for manipulating KDB or Metadata, libease which contains convenience functions and several others.

The Elektra core library, which will be the main subject of my research, contains several modules that provide different functionality. The most interesting for my research are:

- Key - used for handling the lifecycle of a Key (creation / deletion/ ...)
- KDB - used for accessing the included Key database
- Meta - used for reading or manipulating the metadata of a Key
- Value - used for reading or manipulating the value of a Key
- Name - used for reading or manipulating the name of a Key
- Tests - used for 'testing' Keys (comparison / checking the type of Key / ...)
- KeySet - used for handling sets of Keys

3.1.1 Design Goals

The Elektra initiative has a set of stated design goals that are ranked in order of preference. This means that goals that are higher on this list have higher precedence over goals ranked lower. The goals are closely related to the different types of users that can utilize Elektra: Developers, Administrators and Maintainers of Elektra. The reviews will be done mostly from the point of view of developers, since the software API is mainly used by developers.

Stability

Elektra states that within a major release X the API can only be extended, no functions will be removed. This guarantees that any program that has been compiled with an older version of the same major version X still works with the new version as long as the major version stays the same. Those guarantees not only hold for the API but also the ABI. Stability is the most important design goal of Elektra.

Simplicity

Elektra should, in its essence, be a simple key-value store for accessing and persisting key-value pairs. Systems that are too complicated, can cause problems with the usability of the API which in turn leads to reduced adoption. It should be very straightforward for developers to use and integrate Elektra within their applications. Functions should not be overly complicated, and the API should focus on providing basic operations to the programmer.

Robustness

Configuration management with Elektra should be robust, which means that Elektra aims to eliminate a multitude of problems connected with existing configuration management solutions. This includes problems with different behavior on different systems, validation, sensible error messages and much more. Elektra should be stable, easy-to-use and catch errors with configuration settings in advance.

Extensibility

Elektra should also be easily extensible. There are multiple ways to add functionality to Elektra: backend plugins, frontends, bindings for different programming language. Elektra aims to be extensible in every single aspect, and its current architecture reflects that: Many parts of Elektra are implemented as plugins which can be activated/deactivated in a flexible manner. Only a small subset of Elektra is the actual core, which needs to be present on every system.

Performance

Lastly, performance is also one of the main design goals of Elektra, although it is the least important of them. This doesn't mean that it is unimportant to Elektra though, since Elektra's performance has impact on the startup-time of applications and even whole systems. Elektra needs to be of at least comparable speed to existing applications in order to be able to be used as a replacement for existing configuration management solutions.

3.2 Reviewing the API

For evaluating the three review methods I need a subject on which I can perform the API review. This should be an API that is already used in real-world projects and has not recently been the subject of an API review. Otherwise, the results of this case study would be skewed, since I would be reviewing an API that has already been reviewed and reengineered on the basis of those reviews. Additionally, the API should not be developed in any way by myself, in order to avoid potential biases during the review process.

Elektra is a software library which is actively used by other software developers. Its code is open-source and publicly available to study. Currently, the API is in version 0.9, which means that it is still undergoing development and is subject to changes. The API is now at a point where the developers consider stabilizing the public API. Before the developers make that commitment, they want to perform a thorough review of the usability of the API in order to find and fix potential usability issues before they get cemented in by stabilizing the API. This makes Elektra a good candidate for the API reviews, since the API is mature enough and has already seen some real-world use, but still in a state where changes to the API are relatively easy. It has never been reviewed before, so there are no previous instances of reengineering the API that could skew the results of the case study.

During the case study I will perform an API review of the Elektra core API using three different methods: An analytics-based, an empirics-based and my newly proposed reengineering-based review technique. The first two reviews will provide a baseline to which the reengineering-based review will be compared against. The goal of the case study is to investigate whether the reengineering-based approach can be used successfully to review the usability of a software API. I want to investigate the strengths and weaknesses of the review method when compared to the other two types of review methods. After all three reviews have been conducted, I will use the data gathered during the review processes to provide an account of the viability of the reengineering-based review method. I will also perform a qualitative analysis of the strengths and weaknesses of the reengineering-based method that emerged during the case study.

3.2.1 Research Questions

The results from the case study and the following analysis of the performance of the reengineering-based review method will be used to answer the following research questions:

1. Does the reengineering-based approach find usability issues that have not been found by the other approaches?
2. Does the reengineering-based approach find different types of usability issues compared to the other approaches?
3. Are any of the usability issues found by the reengineering-based approach related to language-specific behavior? Do the other approaches find any issues with reliance on language-specific behavior?
4. What are the general up- and downsides of the reengineering-based approach compared to the other approaches?

3.3 Methods

3.3.1 Analytic

The first review method used is an analytics-based approach for identifying issues with a software API. This method works by using a set of API design principles and applying those rules to the existing software API definitions. By identifying discrepancies between the public API and the guidelines, problems with the API are uncovered.

Review Guidelines

A set of guidelines, which are compiled beforehand, form the basis upon which the review is performed. To build this set, I analyzed multiple existing academic works about analytics-based reviews and the guidelines used in those reviews. There is already extensive work on this topic, which led to the discovery of exhaustive lists of guidelines for software API design, such as the ones by Mosqueira et al. [MRARMB+18] as well as Piccioni et al. [PFM13]. The Elektra project itself also already has several documents containing guidelines for API reviews [Ini23b].

Based on those lists, I compiled a checklist (found in Appendix A.1) which I will be using for the API review. Thischecklists merge the selected guidelines from the literature with the pre-existing guidelines of Elektra.

A first step in filtering the guidelines from my literature research was checking whether they were applicable to Elektra. Many of the guidelines were intended for use with object-oriented programming languages. A large part of the reviewed code base is written in C, which is not an object-oriented language, so I was searching for more generally applicable guidelines as well as guidelines specifically intended for reviewing C applications.

Another good orientation for deciding which guidelines to include was a study performed by Zibran et al. [ZER11], which categorizes API usability issues and evaluates how often certain categories of issues occur. This helps select guidelines concerning more common problems. The study also includes a list of usability factors coupled with descriptions. Those guidelines were included in the large collection of guidelines I used as a basis for creating the final checklists.

In addition to the resources mentioned above, another important resource were the already existing guidelines of Elektra. In the code repository of Elektra there are different documents containing guidelines for developers. Those guidelines contain best practices related to code style, API design and software architecture. The existing guidelines have also been incorporated into the checklists for the reviews.

The guidelines included in the list are sorted by their respective categories, partly based on how Zibran et al. [ZER11] describe them: Documentation, Naming, Parameter & Return Types, Structural Clarity, Memory Management, Extensibility, Tests and Other. This enables the categorization and quantitative analysis of the uncovered issues for further evaluation - as described in the results section.

I summarized and merged the lists the literature research yielded into a new checklist that will be used for conducting the usability study of the Elektra API. The checklist contains heuristics specific to single functions, which can be applied to singular functions in an interface.

The bullet points on the checklist were designed to be easily answerable with either fulfilled, unfulfilled or not applicable. If answers to a bullet point would require a more elaborate explanation than just one or two sentences, merely checking the bullet point is not enough. I tried to avoid those situations by further simplifying and splitting some bullet points from the initial list I compiled. The end goal of that refinement step was that the reviewer would be able to judge the status of each bullet point quickly, without the reviewer having to write a long summary / explanation for every item on the checklist. This eases the review process for the reviewer as well as the moderator.

Review Process

After researching and comparing several processes for performing a review, I settled on a review process similar to the one described in [MRARMB+18]. This review process works by creating checklists and then reviewing the API with the help of those checklists. In the review process each function is reviewed separately. The process for this is illustrated in the figure 3.1.

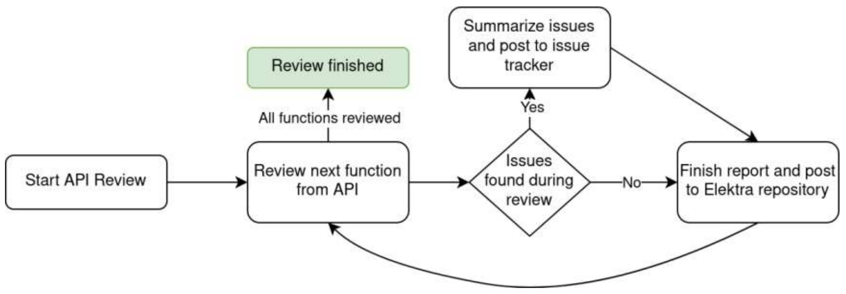
Summarize issues and post to issue tracker
Review finished
Yes
All functions reviewed
Start API Review
Review next function from API
Issues found during review
No
Finish report and post to Elektra repository

Figure 3.1: Process of API review for the analytics-based review method

The reviewer checks during a function review whether the function fulfills the guidelines from the checklist one-by-one. This process is illustrated in figure 3.2 If any issues are found, those issues will be documented for later discussion by Elektra developers. It is not the goal of the review process to find solutions for those issues, but rather to uncover any issues and report them in the Elektra issue tracker. Solutions for uncovered problems will be created at a later point in time, the goal of the review is merely uncovering as many issues as possible.

The basic idea behind this review method is: Break the review down into many small bits, that can be performed easily and alone in short sessions. The guidelines are broken down into small bits, the interfaces are broken down into small bits (their functions). This makes it easy to perform the review systematically, covering every function and every guideline.

The part of the code base, which will be reviewed, is the Elektra Core API. The complete interface can be found in the header file kdb.h [Ini23b]. Every function in this file will be checked with the help of the checklists. I will evaluate whether the function violates any of the guidelines from the checklist. If it does, then that point will be marked as 'unfulfilled'. Otherwise, this point will be marked as 'fulfilled'. If a bullet point is not applicable to a function, then it will be marked as 'not applicable'. A short reason must be given, for why the reviewer feels the guidelines is not applicable, if it is not immediately clear.

To document the review process, a short summary for each review of a function is created. This summary includes the following information about the review:

- Time of the review (Start and End)

- List of Guidelines and their status (fulfilled, unfulfilled, not applicable)
- List of issues generated (title, description, type, github link)

Every violated guideline constitutes a potential issue with the Elektra API. If the issue is more complicated, and its solution requires some discussion and bigger changes to the codebase, the reviewer summarizes the found issue and explains the problems found during the review process.

Afterwards, the issue, together with the remarks of the reviewer, is posted to the issue tracker for further discussion. The Elektra organization gathers input from the core developers on the different issues and proposes solutions for the uncovered issues. This step is essential to the review process, as the created issue is only the starting point for further discussion. The intention is to get a discussion going that leads to a satisfying solution for the uncovered problem. The findings of those discussions and a description of the solution will then be added to the review summary as well, to help with the evaluation of the review method.

Many analytics-based review methods such as the Apiness review method proposed by Macvean et al. [MMD16] use multiple reviewers for the same function, since opinions can differ widely between reviewers. This leads to a significantly higher probability of discovering possible issues with the API. To get better results, input from different viewpoints is needed. I will perform some reviews of the public API with members of the Elektra community as well. Sadly, due to resource constraints, I will not be able to conduct reviews of all functions with multiple reviewers.

An overview of the function review process can be found in the figure 3.2

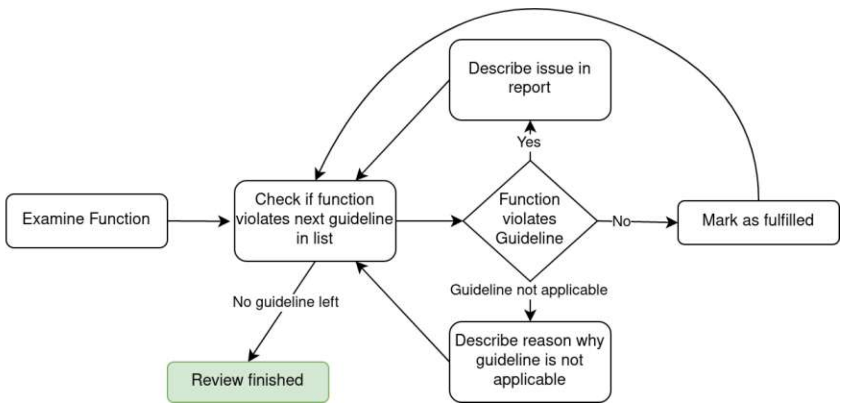
Examine Function
Check if function violates next guideline in list
Function violates Guideline
Yes
Describe issue in report
No
Mark as fulfilled
Guideline not applicable
Describe reason why guideline is not applicable
No guideline left
Review finished

Figure 3.2: Process of review for a single function during the analytics-based review

The peer-review sessions will include two people, one moderator and one reviewer and have a similar structure as the Apiness review method introduced by Macvean et al. [MMD16]. I will assume the role of the moderator, while the respective Elektra community member assumes the role of the reviewer. Those reviews will be performed before I review the rest of the API alone. This helps me get a better insight into different people's viewpoints and interpretation of the principles contained in the checklist. The learnings from performing those reviews will further help me in conducting solo reviews of the API.

In the peer-review sessions, the reviewer is given the function signature, the function's documentation, its test code, as well as the checklist by the moderator. The moderator gives a short introduction to the review process and answers any questions about the process that might arise. During the review, the moderator first reads out aloud the currently discussed checkpoint. Afterwards the reviewer and the moderator discuss whether this particular bullet point is violated in any way. The moderator also simultaneously creates the documentation for the review, in the exact same fashion as in the solo reviews.

Optimally, the reviewer should not be the author of the respective method in the public interface, to avoid any bias. Based on that, the functions that a reviewer reviews during this process are chosen by the moderator in advance, in order to avoid any such clashes.

Scope of the Review

For the C-API I will review all 7 modules of the Elektra core API: Key, KeyTest, Meta, Name Manipulation, Value Manipulation, KeySet, KDB. All functions of this module included in the public API will be part of the review. Private functions are excluded since they are not considered part of the public API.

3.3.2 Empirical

The second review consists of a review of existing code bases utilizing Elektra for configuration management, based on a review process proposed by [Fag02]. The review will be conducted on the KConfig [KDE23] and GSettings [Fou23] backend implementations of Elektra, which are contained in the Elektra source code repository [Ini23b]. While I utilized some help from additional reviewers in the first review, this time I will solely rely on myself as a reviewer.

KConfig and GSettings

KConfig and GSettings are both APIs for storage backends for storing configuration in the KDE / GNOME desktop environments respectively. Elektra provides implementations for storage backends for both of those APIs. The empirics-based review will be conducted on the implementations of the Elektra storage backends for KDE and GNOME.

The configuration needs of desktop environments are usually very broad, because the configuration encompasses a wide range of settings - which can contain a lot of different kinds of values. They also usually have to manage configuration for multiple users, which in turn makes extensive use of key hierarchies. Additionally, it also makes use of many advanced features provided by the Elektra API, such as key locking. At last, both backends also make use of all the different structs Elektra provides - Key, KeySet and KDB, thus covering a wide range of usage scenarios. The fact that both backends cover a large amount of usage scenarios is important for the review, so I can cover large parts of the Elektra API with me review.

KConfig itself is written in C++, while Elektra is a C library. This KConfig implementation uses the C++ bindings of Elektra, which are not necessarily directly translatable to the respective Elektra C API functions. Larger projects usually write their own small set of wrappers / facades to consume external APIs. The KConfig storage backend implementation has its own set of wrappers for Elektra, so I will have to identify usages of the wrapper functions in the source code and trace them back to their respective Elektra API invocations.

GSettings, on the other hand, is implemented directly in C, which is the language in which the core implementation of Elektra has been written as well. In the review included are the glib bindings for Elektra as well, because the main usage of the Elektra API is contained in the source code for the glib-bindings. The glib source code is needed to track down the specific invocations of functions from the Elektra public API.

Goal of the review

The goal of this review is to uncover misuses of the API by the implementor, which gives hints about problems with the usability of Elektra's API. By examining an existing application using Elektra for configuration management, it is possible to see how the API is used by developers in a real life scenario. This review process is similar to a standard code review, but it will be more thorough and I will be looking specifically for misuses of the Elektra API, no other issues with the existing code bases.

One example for misuses I am looking for, would be storing string values as binary Key or using 'wrong' functions to achieve a specific goal. Other misuses would be unnecessary chains of functions that could be replaced by a simpler chain of function calls or even a singular function calls. Any problems not specifically related to the usage of the Elektra API are of no interest for this review and therefore ignored.

Review Process

The empirical review consists of a code inspection. I will be performing the review on code of developers utilizing the Elektra API. Fagan [Fag02] describes a methodology for performing code inspections that consists of 5 steps: Overview, Preparation, Inspection, Rework and Follow-Up. I will be basing my code inspection on those 5 phase, although I will be adapting them as described below to account for the fact that I am performing those reviews alone.

Additionally, a common tool for aiding in performing code inspections are checklists such as proposed by Brykczynski [Bry99]. During the code inspection I will be using a checklist (see 3.3.2) that is specifically created for this use case. Since I am not reviewing the code itself, but use the code review to gather information about how the API is used by developer, I will have to create my own checklist specifically tailored for that use case instead of relying on existing ones. Ususally, the items contained in checklists for code inspections focus on finding issues with the code that is being reviewed, rather than with the API that the code is using. This is why I will create my own checklist specifically tailored to finding issues with the usability of the API that gets used within the reviewed code.

Overview I will start the review process by finding all usages of the Elektra API in the respective implementations. After getting a good overview of where and how the Elektra API is used, I specifically look at every invocation of an Elektra function in the code to check if they are used in a correct manner. The surrounding context will also be considered in order to detect any violations of Pre- or Postconditions specified in the documentation of the functions. Any problems with the usability of the Elektra API that surface during this review process will be documented by me for later inspection, similar to the analytics-based review. This process is illustrated in the figure 3.3

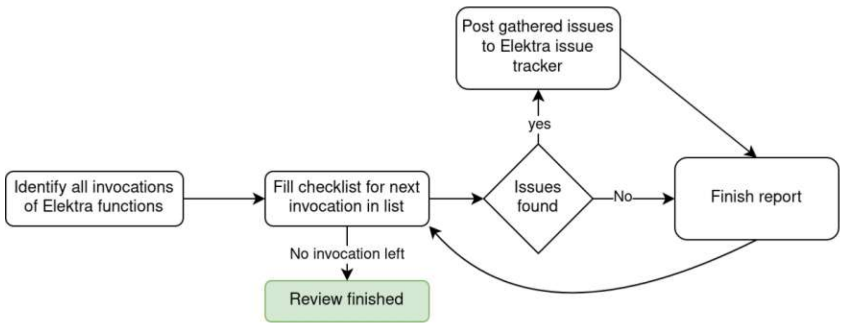
Post gathered issues to Elektra issue tracker
yes
Identify all invocations of Elektra functions
Fill checklist for next invocation in list
Issues found
No
Finish report
No invocation left
Review finished

Figure 3.3: Review process for the empirics-based API review

Issues that occur repeatedly will be documented in one place, only referring to all the parts of the code where the API has been misused that way. The amount of times an issue occurs can give an indication on the severity / importance of the issue. Frequent misuses might indicate a bigger problem, which in turn tends to be more important to fix.

One problem with this method is, that the KConfig code has mainly been written by one developer that contributed 60% of the whole source code. The other 40% are split between numerous other developers, 83 in total.

The GSettings backend has been mainly written by 2 different developers, where one delivered a basic backend, which has been overhauled by another. Minor changes and adjustments have been made by other developers, that consist mainly of bug fixes.

This can skew the results, because one developer might have had a hard time understanding a specific concept, while finding it easy to understand another one. If many mistakes stem from a specific misunderstanding, then the results will heavily skew in towards those specific issues. Other developers might struggle with different concepts or functions, that other developers didn't. Those issues might be overlooked in this review.

Using 2 different projects, that have been authored by different developers, should alleviate this issue. There have been 4 different main developers involved in the creation of the two projects, with some minor contributions from other developers. While this number could certainly be higher, this should prevent this review from skewing heavily towards the issues encountered by one specific developer.

Preparation In order to get an overview about all the usages of each function contained in the Elektra public API, I will use a checklist for each function, similar to the checklists used in the methodology of the analytics-based analysis. Just like in the analytics-based analysis I create a log of the review containing the checklist for every function in the Elektra public API.

This time I will not check for the fulfillment of every bullet point, but rather collect all the information pertaining to a specific function of an Elektra API. I document every usage of the function I encountered and reviewed. Additionally, I collect all uncovered problems, that are related to a specific question in the checklist, in the review log when I encounter them during the review. At the end of the review, this should give me an overview of all the issues related to a specific function.

The template for the checklist used in the review looks as follows:

- How often is the function used?
- Are there any examples of wrong usages of this function
- Is the usage of this function minimal? (e.g. could another function be used to achieve the same functionality in a more straightforward way)
- Does the code adhere to the pre/post-conditions and invariants given by the documentation of the function?

I want to achieve several goals with this selection of questions:

The first question answers how important this function is for users of the API. Functions that get used a lot are very important, while functions that are never used are an indicator for superfluous functions, which might be candidates for removal. On the other hand, the C++ bindings provide some additional functionality utilizing C++'s object-oriented paradigms. Functions that are provided via this way that are used often, could give an indication to add similar functionality to Elektra's core API.

Wrong usages of functions indicate a problem with the documentation or general design of the function. Looking at commonly misused functions, one can think of improvements to make those functions more intuitive. For some use cases, there are several ways on how to achieve a certain functionality, but preferably there should only be one way to solve a certain problem using the Elektra API. Using unsuited functions for certain use cases can give indications about problems with the discoverability / intuitiveness of the Elektra API.

Inspection The review itself is conducted by looking at the source codes of both backend implementations and identifying all usages of the Elektra API, and then using the checklist to find issues with the Elektra public API.

The GSettings backend uses a layer in between the Elektra public API and the implementation of the backend. These are called the glib bindings. Because all functions in this layer are prefixed with gelektra, it is very easy to find all usages of Elektra functions in the GSettings backend via the following command:

```
grep -E "gelektra_[A-Za-z\-_]*" -o elektrasettingsbackend.c
```

Listing 3.1: grep command for extracting all gelektra function invocations from GSettings backend

For creating a statistic about the amount of invocations of functions in the GSettings backend I used the following CLI command on Linux. The output of this command can be found below in the respective results section 3.4.2:

```
grep -E "gelektra_[A-Za-z\-_]*" -o elektrasettingsbackend.c | sort | uniq -c
    | sort -r
```

Listing 3.2: CLI command for extracting the count of all gelektra function invocations in GSettings backend

For the review of KConfig I manually parsed through the source code, as it seemed unpractical to write an all-encompassing grep statement. This manual parsing has been used to gather information about all the invocations of Elektra public API functions. This is due to the nature of C++, where methods can be called on objects. I would have to find every variable instantiated with an Elektra object and then finding all method invocations on the object.

Because the source file of the KConfig backend is relatively short anyway, manually sifting through the source code is a quicker way of compiling a list containing all invocations of functions using the Elektra API than trying to automate this task. Because of this, I will also translate the method invocations to their respective C API function, if there is a corresponding function in the C API. This should be the case for most function invocations in the backend.

For the count of the usages of the different functions in the KConfig backend I compiled a manual list (function_invocations.txt) of all occurrences of the functions and piped it to the same command as for the GSettings backend above, only removing the grep part and passing the list of method names to sort directly:

```
sort function_invocations.txt | uniq -c | sort -r
```

Listing 3.3: CLI command for counting all functions occurences in KConfig

The Elektra core API contains functions, that take an options argument. Those can alter the behavior of the function in question drastically, one instance being ksLookupByName, which does different things with the found Key - ranging from deleting the Key, to doing nothing at all. I will look at the usage of the different flags of the functions as well in order to discern their respective use-cases.

Another interesting source of information for this review is the git history of the reviewed source code, as already shown by Murphy-Hill et al [MHSH+18]. By searching through the git history as a whole and systematically (using keywords like bug, error, fix, ...), additional misuses of the API can be found and included into the analysis as well. This gives important information about the evolution of the source code and any errors the developer might have corrected already. The results from this historic search will be treated in the same way as the issues found in the current version of the source code. Functions whose signatures have changed drastically, since the respective commit will not be included in this step.

For searching through the history, I look at the changes introduced in the specific commits, rather than using grep for searching through the whole source code again. This should be the faster method of evaluating the historic results, as I don't have to skip through a lot of duplicate occurrences for every commit. In cases of big diffs, I might use grep to search through those diffs individually. This case should be more of an exception than the norm, usually just looking at the changes introduced in the respective commit will suffice.

Rework After all parts of the source code, as well as all historic changes, have been reviewed and all issues documented within the review log, I will evaluate the results of the review. First I will compare the amount of invocations each function in the API has and compare them with the amount of other functions. This gives an overview of which functions are the most relevant, and which functions are rarely used. This can give indications about the minimality of the API. If there is a large amount of unused functions, the Elektra developer team might consider cutting some from the public API.

Some functions might provide a shortcut function, whose functionality can otherwise only be accessed by using several functions from the API in a chain. During the review I already try to look out for such function call chains, that could be replaced by an existing convenience function. Additionally, if there are some function call chains that do not have an equivalent shortcut function, it might prove worthwhile to add such a shortcut function, in order to make code utilizing the Elektra API less verbose. This has to be done after careful consideration though, as this runs contrary to another stated goal of the API - simplicity (as explained above).

The fourth question in the review checklist is related to the contracts of the functions as stated in the documentation. By analyzing the context of the function invocations I hope to find problems with the contract of the function. If there are function invocations where the context violates the contract of the invoked function, then the contract needs to be adjusted to explicitly include / exclude this usage, depending on the context and function. Especially the historic data can give important information about initial misuses of the

API, which might stem from missing / wrong pre/postconditions in the Elektra API documentation.

Follow-Up After I finish the evaluation and compile a list of uncovered problems, I will write a short summary about the respective issues and propose one or several ways to fix the problem. Afterwards I will post each issue from the review to the Elektra issue tracker with the summary and the proposed solution(s). In the bug tracker the developer community of Elektra discusses and assesses the proposal. After the developers of Elektra acknowledge it as an issue with the API as well, I will discuss the possible solutions. After agreeing to a solution, I will then implement it in the Elektra source code.

3.3.3 Reengineering

As the third and final method for reviewing the Elektra API I will use a method based on reengineering the existing C-API in the Rust programming language. For this review I will reengineer the Elektra core API in Rust and provide a C-FFI compatible interfaces to the existing C-API.

The main reason for reengineering the API in a different language is that the Elektra API is intended as a standard for configuration management. The current implementation should only be considered a reference implementation of that standard. The API is not part of that standard, a reference implementation only needs to provide the same functionality and behavior. Using a different language enables me to look at the existing standard from a different angle, which in turn might lead to discovering better solutions for the design of the existing API.

During the process of reengineering, the reviewer has to analyze every component and function contained in the old API. This should lead to the reviewer discovering problems with the design of the current API and coming up with alternative solutions for the design of the API. The scope of the review is once again the Elektra Core API as described in 3.3.1.

Review Process

As a basis for the reengineering process of this review I will use a reengineering approach as described by Rosenberg et al [RH96]. Figure 3.4 shows the general process for reengineering software as proposed by Rosenberg et al.

Rosenberg et al [RH96] describe five phases for a reengineering development project. I will also use those phases during my reengineering process, albeit I will skip the 'Re-engineering Team Formation' and 'Project Feasibility Analysis' phases.

The team for the reengineering process only consists of me, since I will be conducting the case study. I will also consult the maintainers of the Elektra during the 'Analysis and Planning' process in order to gain insights into the functionality and requirements of the existing software API.

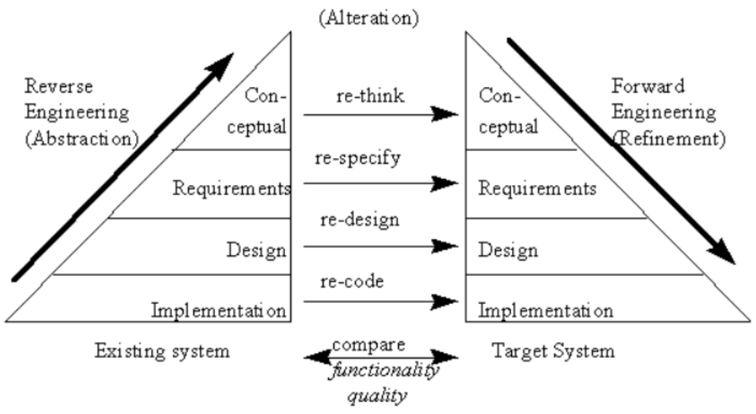

Figure 3.4: General Model for Software Re-Engineering (taken from [RH96])

The 'Project Feasibility Analysis' phase is not important for my case study, since it mainly serves economical purposes which are not relevant for this case study. Quite to the contrary, the case study is a feasibility analysis in itself, since I want to prove the feasibility of applying the reengineering process as a review method for software APIs.

This means the reengineering review process will consist of three phases:

- Analysis and Planning
- Re-engineering implementation
- Testing and transition

An overview of the process for the reengineering during my review can be found in figure 3.5

Analysis and Planning The 'Analysis and Planning' phase consists of three sub-phases: Analyzing the legacy system, specifying the characteristics of the target system and creating a validation suite for validating the correct transfer of functionality.

The basis for the analyzing the legacy system are the Elektra header files, which describe the current state of the API, as well as the whole Elektra documentation. Since Elektra is intended as a standard there is already comprehensive documentation on requirements and use-cases for an implementation of that standard. Because of this, the existing documentation already describes the functionality and the desired properties of an implementation in great detail. If any additional questions arise during this process, I will consult the Elektra maintainers to get insights into certain functionality. This will also help improve the current standard.

The desired characteristics of the target system align with the design goals of Elektra stated in section 3.1.1. The new API should fulfill those requirements at least as well as the existing API, but make improvements to the fulfillment of those design goals wherever possible. In addition I will be placing a special emphasis on improving the usability of the reengineered API. In order to achieve this I will additionally incorporate goals defined by Tahvildari et al [TKM03] in my reengineering process. As already mentioned in the Background section 2.3.3, 'High Code Naming and Commenting Quality' as well as 'High Structure Quality' are particularly factors related to the usability of a software API.

One of the requirements of the reengineered API is that it exposes an interface that is identical to the current API. I will use this fact for the validation phase. The resulting shared library should be able to be linked against existing programs utilizing Elektra. This enables me to use the existing test suite for the Elektra Core API. This helps in ensuring that the new implementation is behaving exactly as the old library and enables me to draw from the already existing and comprehensive test suite. Using the existing, comprehensive test suite should enable me to verify that the reengineered API is working as intended.

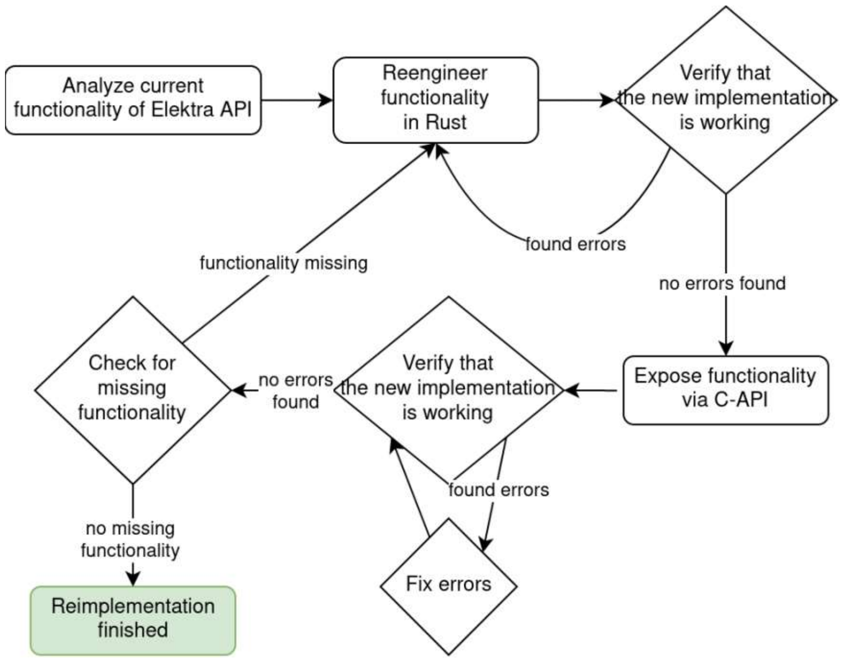
Analyze current functionality of Elektra API
Reengineer functionality in Rust
Verify that the new implementation is working
found errors
no errors found
Expose functionality via C-API
Verify that the new implementation is working
found errors
Fix errors
no errors found
Check for missing functionality
functionality missing
no missing functionality
Reimplementation finished

Figure 3.5: Reengineering Process for the reengineering-based API review

Re-engineering Implementation This corresponds exactly to the normal software development process. I will create a design based on the analysis created in the 'Analysis and Planning' phase of the reengineering process. After I have settled on a design, I will implement the proposed design in the Rust programming language. The goal is to create an alternative implementation of Elektra that has the exact same functionality, without adding or removing any functionality.

Testing and Transition As already described in the 'Analysis and Planning' 3.3.3 section I will use the existing test suite for validating the functionality of the reengineered software system.

The transition process will be different from the usual transition phase, since the reengineered system is not intended to replace the old system as a whole. The goal is to use the reengineered API to improve the existing API. After creating and testing the reengineered API, the transition phase will consist of comparing the new API to the old API and looking at the parts where the APIs differ. This comparison will expose problems with the usability of the current API and I will create proposals to fix the problems in the existing API using the designs obtained from the reengineering process. In this phase I will also create a qualitative analysis of the issues found throughout the reengineering process.

3.4 Results

3.4.1 Analytic

The amount of issues found during the analytics-based review is almost exactly 1000 (1007), although most of them are minor issues such as typos / minor inaccuracies in the documentation. Those issues are distributed as following between the different categories:

- 690 Documentation
- 92 Naming
- 6 Parameter & Return Types
- 28 Structural Clarity
- 2 Memory Management
- 13 Extensibility
- 175 Tests
- 4 Other

Those issues include every minor and major issue found, there has been no differentiation with regard to severity or scale. A simple typo or a major issue with memory management both count as one issue for this analysis. This explains the strong overrepresentation of the documentation issues, as many of the issues found were minor problems with the documentation, where even a typo counts as one issue.

Many of the found issues within the documentation category refer to the complete absence of pre- and postconditions or invariants or incomplete contracts. Out of the 690 issues regarding documentation, 202 were about missing or incomplete preconditions (68 / 73 functions), postconditions (66 / 73) or invariants (68 / 73).

In the following sections I give a short overview and qualitative analysis of the core issues found in the Elektra core API during the analytics-based review. The qualitative analysis should give a better overview of the most important pain points discovered in the analytics-based review than the short quantitative analysis above.

This analysis includes systemic issues found within the Elektra API, that require broad, sometimes breaking, changes. Finding a solution for those issues also requires the involvement of the whole Elektra development team and discussing possible solutions in the issue tracker before settling on how to fix the issues uncovered.

Inconsistent or Missing Return Codes for Errors

When using the C-API of Elektra it is currently very hard to differentiate types of errors. For instance, the function keyGetBinary has several error-cases which all return -1:

- on NULL pointers
- if maxSize is 0
- if maxSize is too small for string
- if maxSize is larger than SSIZE_MAX
- on type mismatch: binary expected, but found string

This makes it hard to differentiate different types of errors, which in turn makes it hard to generate proper error messages in Elektra's bindings. Those error messages would prove helpful for users of the API or its bindings. In the above case only a generic error message can be generated and the developer using Elektra has to go through every possible error and check what kind of error he made. Another possibility would be for authors of the binding to check for each error case described in the documentation manually and then emitting a fitting error message. This can be very cumbersome to implement for every binding and might be better solved by Elektra providing a built-in mechanism for handling those errors.

Without debugging or explicit checks, users of the API don't know whether there are issues with the maxSize parameters, or type mismatch or simply NULL pointers in the case of the keyGetBinary function. Bindings could check for different error codes and generate different error messages for the different types of errors, if the function returned different error codes for different errors. For instance, if the user passed a wrong maxSize parameter the returned error code could be different than if the user tried to get binary data from a string type Key.

In case of an error, those functions can only return a NULL pointer. This would make it impossible to distinguish between different kinds of errors. Some functions try to alleviate this issue by passing an error Key parameter explicitly. In case of an error, the error Key contains additional information about errors that occurred during the function call. This introduces additional work for the user of the API, as he has to create that error Key and pass it to the function every time it is called. He also needs to check the content of the error Key, which could in turn cause errors in and of itself. The user would also have to take care of handling those errors.

A more convenient solution for this would be creating a global error variable which can be read in case of an error. This has several upsides: The user doesn't have to create an error Key every time he calls a function that returns a pointer. This makes calling the functions more convenient for the user, as he doesn't have to manage the error keys required. It also reduces the size of the signature of the function, which makes understanding and calling them easier for new users of the library. Furthermore, it also creates a centralized way of storing and reading error messages that stem from the usage of the API. This error messages could then be used in the bindings and to unify the error messages across the different bindings.

Preconditions / Invariants / Postconditions

Design by contract [Mey92] is a concept that tries to ensure interoperability between different parts of a program by creating contracts that specify and guarantee how functions will behave. Adding pre- and postconditions as well as invariants was a stated goal by the Elektra development team, so checking for their existence has been included in the checklist for the API review.

Only very few of the reviewed functions already had such conditions included into their documentation. The functions that already had a contract usually only had either pre-, postconditions or invariants but not all three. In most cases, if such conditions existed, the existing ones also needed to be extended with further conditions / invariants.

In the aftermath of the review those contracts have been added to all functions where it was deemed suitable, improving the usability of the API by providing safe guarantees for the users.

Memory Management of functions copying values

The Elektra Core API contains several functions that copy data into a user-supplied buffer. While this allows for a great deal of flexibility for users of the API, this flexibility is often not needed and forces the user to write lots of boilerplate code. This impacts usability of the API and is a source of many possible errors, as users need to constantly check the size of every value they want to copy. This makes using the Elektra API very verbose when copying specific values.

An example for such a function would be keyGetName():

```
ssize_t keyGetName (const Key * key, char * returnedName, size_t maxSize)
```

Listing 3.4: Definition of function keyGetName

This function copies the name from key into a user-supplied buffer returnedName. This function forces mental load onto the user, which shouldn't be necessary to accomplish a relatively simple task. The users of the API need to allocate a new buffer with a proper size, so they need to check the size of the name beforehand via keyGetNameSize(). Afterwards they need to allocate a new buffer and then pass it to keyGetName(), including the maximum size obtained from keyGetNameSize().

In most cases a user just wants to obtain a copy of the name, he doesn't really care about the size of the name or about managing buffers. The function itself could handle that for the API users, leading to a more usable API. Copying into user-created buffers would still be possible in the same way with built-in C functions such as memcpy.

Several other functions in the API operate in exactly the same way as keyGetName() and could be improved in the exact same manner:

- keyGetUnescapedName()
- keyGetBaseName()
- keyGetString()
- keyGetBinary()

The solution proposed for this problem includes removing all the above-mentioned functions and putting descriptions into the documentation urging users to use strncopy or memcpy for those tasks instead. The reasoning is that this reduces the amount of functions in the public API, while retaining all the functionality. Furthermore, using built-in C-functions should be more intuitive to developers of C than using special functions provided by Elektra that essentially call the built-in functions for the users of the API.

keyString() issues

keyString() is a function for getting the value stored in a Key as a string. Because the value of a Key doesn't necessarily need to exist or be a string, there need to be special cases for when the Key's value doesn't represent a string. Currently, the function returns special strings, if that is the case: "(null)" and "(binary)".

There are several reasons why this is suboptimal:

- If a user were to store the value "(null)" for some reason, he could never be sure whether the Key he just queried has the value "(null)" or whether he tried getting a NULL value from a string.
- It is very unintuitive from a user's perspective that string literals are returned for those special / error cases. Many users would not expect this.
- From a puritan's point of view this seems very inelegant, as the function should either return the correct string or have a special way of handling errors. Mixing both can lead to confusion and is definitely not considered best practice.

The solution for this problem I have settled on together with the maintainer of Elektra - after some discussion - changes the return codes of the function in those special cases to return 0. This was deemed the best way to go forward by the development team, as the most intuitive and practical return code. Detailed information about the error that occurred will be instead provided via a second channel, as discussed in the section 3.4.1.

Other minor issues

There have been other minor issues discovered, that are not big enough to deserve their own section. To not leave them out completely I will shortly describe each of them here.

Naming Some functions in the Elektra public API are named in a way that can be confusing for users unfamiliar with the name. Discussions about this topic are often highly subjective by nature, so careful consideration of different opinions was needed to decide on the way forward. During my review I felt that, the functions ksAppend and ksAppendKey in particular have confusing names, which should be changed to better reflect their functionality.

keyCopyMeta Bug The function keyCopyMeta in the Elektra API sets the metadata of one Key to the metadata of another Key. Or at least, that was the intended purpose of this function. Because of an implementation error this function copied all Keys that were contained in the Metadata of the source Key, but Keys in the metadata that existed before were not cleared. This behavior has been confirmed to be a bug and was fixed subsequently.

Split ksLookup functionality ksLookup accepts a second parameter that describes the exact operation that should be performed after the lookup succeeded. Passing KDB_O_POP as a second parameter causes the found Key to be removed from the KeySet and returned. This functionality was deemed important enough to be moved to its own function, ksRemoveKey. This aids discoverability of the API, as such an elementary operation should not be part of another function.

One-line if statements Many of the if-statements that just check for NULL pointers or read-only Keys are written as one-line statements. While this may sound unimportant at first glance, this creates several problems with the Core API. Coverage Reports are hard to parse because of this, as this makes it hard to check whether only the expression in if-statement is evaluated or also the body of the if-statement.

3.4.2 Empirical

In order to evaluate the results of the review, I analyzed the usage of the Elektra Core API in the two implementations of GSettings / KConfig backends. This numbers can give indications about which functions are the most / least used.

GSettings

Because GSettings does not directly invoke the Elektra C API, but uses a thin wrapping layer - the Elektra glib bindings - the function names are the respective glib-binding counterparts of the Elektra functions. Those have the same signature as the underlying Elektra C function and can be considered equal. I added the name of the respective functions from the Elektra public API in parentheses for clarification.

The distribution of function invocations in the GSettings bindings is as follows:

```
8 gelektra_keyset_lookup_byname (ksLookupByName)
8 gelektra_key_new (keyNew)
4 gelektra_keyset_new (ksNew)
4 gelektra_kdb_get (kdbGet)
3 gelektra_keyset_append (ksAppend)
2 gelektra_key_setstring (keySetString)
2 gelektra_keyset_lookup (ksLookup)
2 gelektra_key_name (keyName)
2 gelektra_key_getvaluesize (keyGetValueSize)
2 gelektra_key_getvalue (keyGetValue)
1 gelektra_keyset_at (ksAt)
1 gelektra_key_isbeloworsame (keyIsBelowOrSame)
1 gelektra_key_getstring (keyGetString)
1 gelektra_kdb_set (kdbSet)
1 gelektra_kdb_open (kdbOpen)
1 gelektra_kdb_close (kdbClose)
```

Listing 3.5: Distribution of functions used in GSettings

In total these are 16 functions that have been invoked 43 times in the whole GSettings backend. The backend only uses a small subset of the Elektra Core API, which limits the review of the GSettings backend to those functions. In total the Elektra C Core API contains 73 functions, while 16 of those functions are used in the GSettings backend. This can be an indication that the current API is a bit too cluttered. While GSettings already has a big scope of use cases, only a fraction of functions in the public API are needed for the implementation of the GSettings backend in Elektra.

Unsurprisingly, functions creating new handles are among the most used functions. One of the two most used functions, gelektra_keyset_lookup_byname, is not though. Here I additionally looked at the different flags used in the invocation of the function. All 8 invocations used the KDB_O_NONE flag. There have been plans on removing options from this function altogether, because ksLookupByName currently has too much functionality put into one function. This is a further indicator, that the Elektra team is onto something with the removal of functionality from this function, as only the 'default' option is used in this case. Since there are no default arguments in C, this also makes using the default functionality more verbose and cumbersome, since KDB_O_NONE hast to be explicitly added to every invocation.

KConfig

The review of KConfig paints a similar picture to the GSettings review. In the KConfig there is only a subset of functions that get used, most even get used only once. The distribution can be found as follows:

```
4 keyNew()
4 kdbGet()
3 keyGetString()
3 key.get()
2 key.begin()
2 keyAddBaseName()
1 ksSet()
1 ksNew()
1 ksCut()
1 ksClose()
1 ksAppend()
1 keyIsDirectBelow()
1 keyIsBelowOrSame()
1 keyGetBaseName()
1 key.end()
1 keyDup()
1 kdbSet()
1 kdbOpen()
1 kdbNew()
1 kdbClose()
```

Listing 3.6: Distribution of functions used in KConfig

There are 21 different functions used in 32 different function invocations. Compared to the GSettings backend, this is less than half the amount of function invocations. This supports my general notion after the empirics-based review, where I thought that the KConfig implementation in general appears to be shorter and more concise.

The only functions which cannot be directly translated to functions of the Elektra core API are: key.get(), key.begin(), key.end(). Those are functions for iterating over the name of a Key. key.get() gets the next part of the name, key.begin() gets the beginning of the Key and key.end() gets the last part of the Key. This can be an indication that such a feature would be nice to have in the core API as well. Especially, because those functions are among then only ones used several times. Further discussion of this topic can be found in the Section 3.4.2

Summary

The review uncovered that the backends only use a small subset of functions from the Elektra public API, which was very different from the expecations I had befor the review. GSettings uses 16 distinct functions, while KConfig uses 21 distinct functions. Taken together, both backends use 24 distinct functions from the Elektra public API. This means, that GSettings only uses 3 functions that are not also used in KConfig.

Both backends mainly use very basic Operations from the Elektra public API, like constructors or getter/setters. Additionally, some functions for testing hierarchy (keyIsBelowOrSame() for instance) or the ksLookup functions are used.

Because such a large part of Elektra functions is missing, the scope of the review also shrinks to those used functions, meaning three quarters of functions haven't been really touched by the review. This could be a reason why there aren't that many results compared to the analytics-based review.

One takeaway from this is, that even with basic operations and a short amount of code users of the Elektra library can implement a whole settings backend for a desktop environment, which usually is a lot more work-intensive when comparing to already existing configuration backends for GSettings / KConfig. This is a positive upside, indicating that albeit simple to use, the user can implement a sophisticated configuration storage with few lines of code.

Another takeaway might be, that the Elektra public API is very extensive and might provide too many features to its users. It suggests that some functions contained in the public API are superfluous and could be removed / integrated into other functions. One of the main goals of Elektra is simplicity, according to its stated design goals. This includes providing a simple, easy-to-learn, yet powerful API to software developers.

There are several downsides to having too many functions in the public API:

New users can get overwhelmed with the sheer amount of functions available in the public API. They can get confused on which functions to use for a certain task. They can also be put off learning the API because it seems like a daunting task.

The functionality that some functions provide might overlap, resulting in two functions providing similar functionality to the user. This also constitutes additional maintenance amount for the developers of Elektra. Additionally, duplicated functionality can introduce subtle bugs when one function changes, but developers forget changing the functionality of the other function as well.

Elektra aims to be simple from the start, providing basic but powerful operations to its users. Too many functions contained in the public API runs contrary to that design goal.

Careful consideration about removing functions has to be made, because a function might have a niche use case that is rarely used, but it still might be important to select few users of the library. Some functions also help improve runtime speed of user's programs, because they can access internals which users cannot access directly. So while some tasks could be implemented by invoking and chaining other, basic functions, they might be more inefficient than the functions Elektra provides out-of-the-box for some tasks.

As a result of this, I proposed a review of all existing functions in the Elektra Core API library which examines which functions are still required to be included in the public API. More information about this can be found in the section 3.4.2

Qualitative Analysis

The empirics-based review yielded very few results, compared to the analytics-based review, which yielded an enormous amount of usable issues, albeit mostly small changes. This might be due to several factors:

The reviewed code has been partly written by developers that already had experience working with or on Elektra itself. This experience might explain why there are few misuses of the API, as users already had some familiarity with Elektra.

The reviewed code base utilizes only a fraction of the functions provided by the Elektra API. Most of those functions are very basic operations, like accessing the name of the Key or setting the name of the Key. Those functions usually have a straightforward design, that cannot be iterated upon a lot, as they are very much dictated by their function.

I expected initially that the historic review, where I examine how the code changes over time, would yield more issues than the it did. I assumed that I would find instances of misuses and bugs there, which could have been prevented with a more user-friendly API design. This turned out to be untrue, as the main reason for the changes was depreciation or change of API functions. There were some issues with the code that have been resolved over time but they mainly related to code that does not invoke any Elektra functions.

The reviewed code bases might have been a too small sample size to properly gauge the usage of the Elektra core API. The GSettings and KConfig backend both implement a fully functioning settings backend in a few hundred lines of code each.

This could also be construed as a positive side of Elektra - apparently it is possible to create a backend for storing all settings contained in a desktop environment with very few lines of code, using mostly basic operations from the provided Elektra API. An indicator that the API provides an easy yet still powerful and flexible way, for integrating neatlessly into existing applications. On the other hand this might also be an indicator that the API could be a bit too loaded, which can then prompt a review of the practical usage of the contained functions.

The scope of the empirics-based review was smaller compared to the scope of the analytics-based review. By checking the function invocations contained in the GSettings and KConfig backends, the documentation and tests of the Elektra public API are not reviewed during the process. A lot of the issues from the analytics-based review stemmed from issues in the documentation of the API. The empirics-based review would not find any issues related to documentation.

Still, there are a few general problems with the Elektra public API, which the empirics-based review uncovered, and I want to discuss those problems and their solutions below:

Name part API During the review of the GSettings bindings one piece of code stood out in particular to me because of its verbosity and complexity:

```
gchar * gsettingskeyname = g_strdup (
    g_strstr_len (
        g_strstr_len (gelektra_key_name (gkey), -1, "/") + 1,
        -1,
        "/"
    )
);
```

Listing 3.7: Snippet from GSettings backend that tries to access a part of the Key's name

This piece of code tries to get the name of the Key starting from the second part of the Key path. For instance if the Key's name is system:/hosts/hostname, then this function would return /hostname. This seems like a convoluted way to access parts of the Key's path. It is also hard to handle all the complex corner cases involving escaped symbols and escaped slashes. The rules concerning that are quite complex and very hard to properly implement, which is why users should rely on built-in functionality to handle this for them. Otherwise, users that do not fully understand the intricacies of the escaping done by Elektra could make errors in re-implementing this logic manually, leading to bugs that are hard to spot and fix.

It would make sense to provide functions that handle such corner cases correctly, because the user might not be aware those exist and add bugs to his software. The user also shouldn't necessarily have to concern themselves with handling every possible edge case, this would introduce a lot of mental load for the programmer worsening the usability of the library. As an example, the current proper solution for iterating over a Key's name parts is:

```
// courtesy of @kodebach
// (https://github.com/ElektraInitiative/libelektra/issues/4102#issuecomment
   -945644189)

Key * k = keyNew ("user:/foo/bar/boo", KEY_END);

const char * uname = keyUnescapedName (k);
size_t usize = keyGetUnescapedNameSize (k);

elektraNamespace namespace = uname[0];

// this if won't be necessary with #3902
```

```
if (usize > 3)
{
    for (
        const char * part = uname + 2;
        part < uname + usize;
        part += strlen (part) + 1;
    )
    {
        // use part
    }
}
else
{
    // root key, has no parts
}
```

Listing 3.8: Current idiomatic way of iterating over parts of the Key's name in Elektra

This is a verbose way of accessing the different parts of the Key, that could certainly be designed easier. In the Elektra issue tracker I discussed several ideas for implementing a dedicated API to access parts of the Key with the maintainers. The solution we settled on after some discussion is adding the following function for accessing parts of the Key's name:

```
/*
  precond: currentPart, must be pointer into the unescaped name of k

  returns the first part (after namespace), if currentPart == NULL
  returns NULL, if currentPart is the last part of k
  returns the next part, otherwise
*/
const char * keyGetNextNamePart (Key * k, const char * currentPart);
```

Listing 3.9: Proposed function for accessing parts of the Key's name

This function can then be used as follows to iterate over specific parts of the name:

```
Key * k;
for (
    const char * part = keyGetNextNamePart (k, NULL);
    part != NULL;
    part = keyGetNextNamePart (k, part)
)
{
    // use part
}
```

Listing 3.10: Example for using new proposed function keyGetNextNamePart

From the direct comparison of both snippets it is already quite clear, that this function would constitute a major improvement over the current idiomatic way of iterating over parts of the Key's name. This reduces verbosity and complexity of the code significantly resulting in easier usability and better discoverability.

There are several use cases for retrieving parts of the name of a Key that can be solved utilizing the proposed API:

- Accessing a part of the Key's name at a specific position
- Accessing a sub path of the Key's name (for instance from part 2 to part 6)
- Iterating over all parts of the Key's name

There is a NameIterator available in the C++-bindings which gets used in the KConfig backend as well. It is even one of the few functions that get invoked multiple times. This API exposes three different functions: begin(), end(), current(), which can be used to access different parts of the name of the string, even iterate over the parts.

This API cannot be ported to the C API though, as it makes use of the object-oriented features of C++. Implementing a similar solution in C would require a lot more boilerplate code, as well as management of internal state. This adds further complexity for users and developers of the API, which should be avoided wherever possible.

While this simplified API can be very helpful in some cases, the empirics-based review also suggests that the API contains too many functions rather than too little. Therefore, it seems unnecessary to introduce another function to the Elektra Core API, as it is unclear whether it represents a minor use case or not. I decided to further research this topic in order to be able to make a well-reasoned decision.

Elektra has a library containing helper functions, which do not rely on any internals to execute their operations. Adding the function discussed above to this helper library seems like the best way to add the feature, as it does not rely on any internals of Elektra. This additionally avoids cluttering the core API.

Additional information about those functions could be included in the documentation of the core API. This would be added to the keyname module, where functions for dealing with the names of Keys are located. This would usually be the first place a user looks for, if he would be searching for functions related to dealing with parts of names. Users might also not be aware of the existence of the Elektra helper library, so pointing them towards that direction also adds significantly to the discoverability of the functionality Elektra provides.

Iterator Functions The inquiry into the possible ways of iterating over the name of a Key also prompted an inquiry into the current method of iterating over the Keys contained in a KeySet. Currently, iterating over Keys in a KeySet is handled by an internal iterator whose state is saved inside the KeySet. This leads to the public API exposing a lot of functions for manipulating the state of the iterators (ksRewind(), ksNext(), ksCurrent()). Additionally, iterating is opaque to the user and dependent on the current state of the iterator, which cannot easily be obtained via the public API.

A Key can also contain a KeySet that holds the metadata for a Key. The API for a Key exposes additional functions for iterating over the metadata KeySet (keyRewindMeta(), keyNextMeta(), keyCurrentMeta()). This clutters the API, since Elektra has to provide functions for every site in the API (Key, KeySet) that is using KeySets. Also, the API for iterating over the metadata of a Key only exposes parts of the functionality that the KeySet provides.

It would make sense to have a common way of iterating over KeySets, which can then be reused across all sites that are using KeySets. This has the upside that this reduces the amount of functions that have to be provided by the API. There only needs to be a way of iterating over KeySets and every part of Elektra, that is using a KeySet, only has to provide a way for the user to obtain that KeySet. The user can then iterate over the KeySet via the common method that is the same for all Keysets. Having such a uniform way of iterating over KeySets reduces the amount of learning for a user, since it is the same for every KeySet.

The current functions contained in the public API for iterating over a KeySet and accessing the respective entries are:

```
int ksRewind (KeySet *ks);
Key *ksNext (KeySet *ks);
Key *ksCurrent (const KeySet *ks);

elektraCursor ksGetCursor (const KeySet *ks);
int ksSetCursor (KeySet *ks, elektraCursor cursor);
Key *ksAtCursor(KeySet *ks, elektraCursor cursor);
```

Listing 3.11: Functions in the KeySet module for iterating over a KeySet

A Key also exposes functions for iterating over its metadata KeySet:

```
int keyRewindMeta (Key *key);
const Key *keyNextMeta (Key *key);
const Key *keyCurrentMeta (const Key *key);
```

Listing 3.12: Functions in the Key module for iterating over the metadata of a KeySet

With the current solution, the user has to know all the functions a specific data type has for manipulating and accessing the iterator. Also, with the current state there are two ways of iterating over the metadata KeySet of a Key. A user could either use the functions provided by the Key itself (which provide less functionality than the functions provided by the KeySet), or he could use the keyMeta() function to obtain a reference to a KeySet and then use the KeySet-specific functions for iterating over that KeySet. One part of the proposed solution is abandoning the functions that a Key provides, since it provides no additional value. A user can just get the metadata KeySet and iterate over it via the functions provided by the KeySet module.

Simplifying this API makes things easier for developers, since functions for manipulating and exposing iterators do not have to be duplicated for every struct that is using a

KeySet internally. With the newly proposed solution new functionality can be used by every site without having to create new functions. This is especially evident with the metadata KeySet, where the Key does not provide the whole functionality for iterating that the KeySet itself provides.

The second proposed improvement for iterating over a KeySet would be to let a user manage the iterator by themselves and only exposing functions for getting the length of a KeySet and for accessing Keys at specific indexes (ksAtCursor()). This is actually already possible with the current public API, but not documented. This has the effect that instead of having to wrestle with the functions that Elektra provides for iterating, a user can just write a simple for-loop. Additionally, this makes iterating over a KeySet feel more natural and analogous to how iterations work in C. The state of the iterator can be managed by the user themselves, which provides additional flexibility.

The current documented way of iterating over a KeySet looks like this:

```
KeySet * ks;
ksRewind();
while (Key *current = ksNext(ks)) {
    // do something
}
```

Listing 3.13: Example for iterating over a KeySet with the initial state of the API

The alternative solution, that is already achievable with the current API, would look something like this.

```
KeySet * ks;
for (int i = 0; i < ksGetSize(ks); i++) {
    Key * current = ksAtCursor(i);
}
```

Listing 3.14: Example for iterating over a KeySet with the newly proposed API

Therefore, removing the functions related to manipulating the internal iterator does not remove any functionality. It makes learning the API easier and reduces the amount of functions in the API without dropping any functionality. Additionally, it ensures that iterating over a KeySet always works the same way and helps new users learning the API not getting confused.

KeyHasMeta A function that has been added to the glib bindings is the gelektra_key_hasmeta function, which returns whether a specific metadata Key is set. Currently, a corresponding function (e.g. keyHasMeta) does not exist in the Elektra core API:

```
gboolean gelektra_key_hasmeta (const GElektraKey * key, const gchar * name)
{
  return (keyValue (keyGetMeta (key->key, name)) != NULL);
}
```

Listing 3.15: gelektra_key_hasmeta function definition

Interestingly enough, while the function is declared in the glib bindings, it is never actually used in the settings backend. This might be an indication that is not needed after all. While having such a function might be convenient at times, the amount of code saved is negligible, which is why I decided against proposing an addition to the Elektra core API.

The review is rather an indication that some functions might not be needed after all, so it is more advisable to reduce the functions available in the public API rather than add even more for miniscule gain.

Amount of functions in Elektra API One thing both backend implementations have in common is, that they use a fraction of all functions available in the Elektra public API. While 73 functions are available, only 24 distinct functions in total have been used across both backends (excluding C++ specific function invocations that have no counterpart in the core API). This can be an indicator that there are too many functions available in the public API.

It is hard to make a definitive statement about the amount of functions contained in the Elektra core API based on the empirics-based review alone. This is due to the fact that the reviewed code bases both have the same, specific use case. While they provide important functionality, the code base itself for the backends is a relatively small sample of code bases utilizing Elektra. There are certainly a broad range of use cases in which Elektra could be used, that differ greatly from the use cases in the reviewed code bases. Nevertheless, this can be an indicator for further research regarding the usage of functions contained in the Elektra core API.

The results of this review suggest that there is a core part of functionality that is paramount to accomplishing the main task of configuration management, while most of the functionality has more of a niche use. On the other hand, even though less people might use some functions available in the public API, this does not automatically indicate that they can be removed or are useless. A trade-off between a so-called 'batteries included' approach and a minimalistic approach has to be made.

As a result of the insights from the empirics-based review, I propose a review of all functions in the Elektra public API, that checks the amount of usage of functions in different code bases. The Elektra code base itself could provide a good starting point for those types of reviews, as the core library itself is used extensively throughout the whole Elektra project.

The proposed review aims to identify functions that are barely used and might be better suited for replacement or merging into different functions. While software developers often think about which additional functions could be added, pruning unnecessary and unused functions is equally important as well and often overlooked. This review method in particular shone a bright light on this issue, where the analytics-based approach seemed to have a blind spot.

3.4.3 Reengineering

Buffers

In the Section 3.4.1 I discussed the up- and downsides of functions that copy into buffers pre-allocated by the caller, compared to functions that simply return a pointer to internal memory already allocated by Elektra. An example for such a pair of functions would be keyName() vs keyGetName(), you can find their signatures in listing 3.16. keyName() simply returns a pointer to Elektra internal memory, while keyGetName copies it into a provided buffer.

```
const char* keyName (const Key *key)
ssize_t keyGetName  (const Key *key, char *returnedName, size_t maxSize)
```

Listing 3.16: keyName vs keyGetName signature

The reengineering-based review provided a very interesting, different angle from the analytics-based review. The initial analytics-based review and the resulting discussion within the Elektra maintainers concluded, that it would be better to only provide one type of function instead of two for every getter. It was decided that all functions taking a pre-allocated buffer as a parameter would be removed. The reengineering-based review showed some problems with the proposed removal of the functions that copy into a pre-allocated buffer in the specific case of implementing a C-FFI compatible interface.

When implementing shared libraries for C in different languages, it is easier to implement an API where the function accepts a pre-allocated buffer. In case of returning a pointer it is hard to avoid memory leaks, as there are only 2 options: The caller has to manually free the returned pointer or call a function that de-allocates the returned pointer for the user of the library. Both possibilities make it hard to keep those shared libraries compatible with either the C-API or existing programs, since such functions do not exist in the Elektra core API and would have to be added. They would only add value in the case of implementations that are not specifically C code.

The new API for the getters in the Elektra core implementation returns pointers to the internally allocated values. The pointers must not be freed or written to under any circumstance, since Elektra handles the memory management. This is part of the function contract, but there is nothing that really enforces those constraints. Users can write to those pointers without any warning from the compiler. Users would have to carefully read the documentation of the function to be aware of this assumption.

In order for the new implementation to be compatible to the existing implementation, the new implementation has to fully handle memory management as well, which is hard given the current state of the Elektra core API.

This also makes it impossible to use newly allocated memory for returning values from the getters, since there is no possibility for the caller to indicate that this memory can be freed. The caller could free the memory himself when it is no longer needed, but this would then again violate the contract of the existing implementation. Depending on the goals of the reengineering process, this can lead to problems, although in most cases it should suffice to simply return the pointer to the memory in the struct in the new implementation.

Issues with the function keyEscapedName There is one getter where this leads to problems, namely the keyEscapedName() function. Since the escaped name is not stored in the struct itself, but rather generated when accessing the getter, it is not possible to simply return a pointer to the internal memory. The escaped name gets generated with the function call and then a pointer to newly allocated memory is returned from the function.

This leads to a problem with the reengineered API in Rust. Since memory has to be newly allocated, it also has to get cleaned up by Elektra at some point. This is only possible if a reference to the memory containing the escaped name is stored somewhere. Currently, the escaped name does not get precomputed and stored in the struct, which means that keyDel cannot properly deallocate the returned string, since it does not know where the newly allocated memory is.

This leads to memory leaks, because the buffer returned by keyEscapedName never gets freed. The user would need to manually free the memory returned by the function. Currently, this is not part of the function contract and would work differently compared to every other getter function leading to confusion and inconsistency in the API.

Generally speaking, one can only safely return values from functions, if the pointer to the data is managed by the respective implementation as well. This is only possible if the pointer to the value is stored somewhere in the Key itself, so the implementation can then later on free the memory when it is required. If this is not the case, then there are several options: The user has to manually free the returned value, the user has to call a function that handles deallocation or the memory never gets freed and is left dangling leading to memory leaks. All of those options are suboptimal.

The reengineering-based review provided interesting additional insights on the matter of which type of getters should be preferred in the API usage. However I decided, that the issues uncovered with the reengineering-based review do not warrant reconsidering our decision from the earlier review, since for the reference C-implementation those issues are irrelevant.

Variadic Arguments

The keyNew and ksNew functions accept variadic arguments in order to create a new Key/KeySet with additional parameters such as name or value or metadata. This aims to provide one single function to create a Key / KeySet. Variadic Arguments are already a bit of a hassle in C, as it is almost impossible to check the number or type of arguments properly. Due to the unsafe nature of this function, this can lead to potential security issues. Also, it is very easy for callers of the function to make errors that are hard to track since they almost always directly result in a segmentation fault. The reason for the segmentation fault might not be immediately obvious, especially for new users of the library. These problems are exacerbated when trying to implement the same behavior in Rust, since Varargs support is not as good as in native C.

```
Key *k1 = keyNew ("user:/example",
    KEY_VALUE, "some data",
    KEY_META, "comment", "comment",
KEY_END);

// versus

Key *k2 = keyNew ("user:/example");
keySetValue ("some data");
keySetMeta (k2, "comment", "comment");
```

Listing 3.17: keyNew variadic arguments versus setters

Since using keyNew in conjunction with varargs does not provide much value over calling setter functions, I propose that it would be better to deprecate and remove the keyNew() function that accepts variadic arguments. In listing 3.17 a comparison of the two different approaches can be found. Instead of using the old function for creating Keys, the user should call the respective setters. The Rust implementation uses a builder pattern for the purpose of instantiating new Keys, which is a good alternative to the variadic argument constructor in C. In C this pattern is hard to emulate due to the missing object-oriented features. There will be no such alternative implemented in C.

My proposed solution for the C library going forward is using the already existing functions for setting properties of Key/KeySet such as keySetValue, keySetMeta and removing the possibility of constructing new objects via the variadic argument constructor.

Key value binary/string split

Currently, there is the option of saving either a binary or a string value in a Key. For this reason there are several functions that operate only on binary/string keys respectively: keyGetBinary, keyGetString, keySetBinary, keySetString. This also further complicates the contract of other functions that deal with values of Keys, for instance keyValue(), which has different runtime behavior depending on the type of the value stored in the key. This clutters the API and makes the behavior of some functions, most notably keyValue(), intransparent and hard to predict for users of the library.

Internally, this split is currently implemented as a union type in C. Since the struct is also considered public API, this change would make the usage of a union type unnecessary. This simplifies the fields of the Key struct as well. Additionally, there is a special type of flag (KEY_NEW) that currently has to be explicitly added to the function invocation when creating a Key with keyNew. This would remove the need for adding this flag to every invocation of keyNew.

My proposed solution is changing the Key to only store binary values internally, since all strings can then be stored in their respective binary representation. The functions related to string values mentioned above would be deprecated and removed. Then there would only be 2 functions left for setting and getting values of a key: keyGetValue() (which is the renamed version of keyValue()) and keySetValue(). This simplifies the API significantly and reduces the mental load on the developers using the library, since they do not have to know the type of the value stored when calling getter functions. keyGetValue() now returns a void pointer to the data stored internally which can be cast to the type that the developer needs. This also enables users of the library to store arbitrary binary data as the value of a Key and then cast it to its proper representation leading to added flexibility when storing arbitrary data as value of a Key. Users could even store structs in the value and then cast them to their proper type, although this is certainly not a recommended use for Key values.

There are downsides to this solution as well, although I think that they are minor in comparison to the upsides. Since keyGetValue() now returns a void pointer, the user has to cast the resulting value to the respective type which can also lead to errors when a value is wrongly cast. Since the main use case for storing values in keys would still be the storage of strings, I do not expect this to lead to big problems for users of the library, since in most cases users will only be storing and retrieving strings. Casts can even be omitted if the Key value is assigned to a variable that already has its type defined, so type casts would not clutter the code of users.

Key locking

Elektra implements a locking mechanism for its Keys. A user can lock any number of the following fields independently: name, value and metadata. When a field has been locked, it can still be read but not modified. This is mainly due to the need for making Keys immutable as soon as they get inserted into a KeySet. Since Elektra does not aim to support concurrent access to Keys, an access model such as a reader-writer lock or mutually exclusive access to fields are not stated goals of the C implementation.

In the standard library Rust implements locking for data mainly through two mechanisms: RwLock and Mutex locks. In order to be able to separately lock name, value and metadata a Mutex or RwLock is required.

Due to the nature of the implementations of RwLock and Mutex, which follow commonly established patterns, the current functionality of Elektra does not fit the usage patterns of RwLock and Mutex. At first glance RwLock seemed like a proper candidate for implementing Elektra's locking mechanisms. But after looking into how locks should exactly work with Elektra, RwLock was deemed unsuitable. RwLock allows either one reference that is writable or an arbitrary amount of reference that are readable. It is not possible to hold a write lock and read at the same time. Therefore when using RwLock it would be impossible to read a field, while it is currently locked for writing. This is currently possible in the C implementation, since a lock in the Elektra core implementation only prevents writing to the respective field, but not reading. A Mutex has the same issue as the RwLock, but here the additional constraint of not being able to hold two read references would hold. This does not make it possible to wrap the respective fields in the struct with Mutex or RwLock for my implementation in Rust to achieve functionality that is equal to the functionality of the C API.

For this reason, I added additional fields for name, value and metadata in the Rust struct that each contain a lock for the respective field. Those are stored separately in the struct, which means they use additional memory for every Key created. This is similar to how the C implementation currently implements locks. This makes the implementation in Rust a bit more cumbersome, since the current idiomatic way of managing access to a struct would be wrapping it or its fields in Mutex or RwLock instances. This would also have the added benefit of more rigorous checking by the compiler.

If Elektra ever tries to implement thread-safe synchronization for accessing different fields, it might be interesting to think about a different approach for locking. Currently, it seems hard to turn the existing lock functionality into a method for synchronizing read and write access to Keys, since this would require changing how KeySet uses the locks. KeySet only requires locking the name of the Key which is why the current solution opted for locking individual fields.

For the future it might be interesting to think about changing how locking works and removing the functionality to lock singular fields. Especially because adding a Key to a KeySet would only require read access when using a RwLock or Mutex approach.

The first benefit would be, that this reduces the complexity of the locking mechanisms. Since a Key would then be able to only be locked as whole or not at all, a user does not have to consider the locks for individual fields when editing multiple fields for example. This also simplifies error handling as well as keeping transactions (editing multiple fields sequentially) atomic. As a tradeoff, it would be possible to use the Elektra locking mechanisms for safely accessing the value of fields in multiple threads.

This would introduce problems with reading the values though, which would be the main downside of this change. When using a read-write or mutex approach for implementing the locking functionality of Elektra it would be impossible to access the fields while they are locked, since both approaches would rule out reading while a write-lock has been acquired.

Structs vs Primitive Types in function parameters & return types

Instead of using primitive types for values or names of Keys, the Rust implementation uses specific structs or aliases for primitive types like KeyName or KeyValue. This makes the signature of the function easier to read and digest for users of the API, since it is now clear what is expected as a function parameter. ElektraName makes it clear that the name of a Key is expected as a parameter, while char* as a parameter type can be ambiguous. Using such structs or aliases throughout the API would make it easier for users to understand what parameters are expected by functions and therefore increase the discoverability of the API. Spiza et al [SH14] even dedicated a whole study to this topic, where they showed that even without type checking type names improve the usability of an API.

CHAPTER 4

Results

4.1 Analysis of the results from the case study

4.1.1 Overview of the review methods

Dimension	Analytic	Empirical	Reengineering
Effort (Preparation)	medium creating a proper checklist can be time-consuming	medium/high finding proper code bases and call sites can be time-consuming	low - high depending on pre-existing documentation
Effort (Review)	low reviews happen quickly	medium parsing through source code can be time-consuming and complicated	high a lot of work for an API review
Amount of issues found	high	low	low
Types of issues found	mostly minor issues, particularly with documentation	minor issues, as well as issues related to function usage	fundamental issues with API design

Analytic

The analytics-based review yielded many minor issues, especially regarding documentation. Additionally, there were many issues regarding naming of functions and parameters, which is also partly related to documentation - since the name of the function has no inherent effect on the technical functionality of Elektra. The analytics-based review also showed systemic issues with documentation. Preconditions, Postconditions and Invariants were only documented sparsely. Many functions were missing all 3 parts of the contract, some of them had only partially documented contracts.

Last but not least, the analytics-based review uncovered some major usability issues which were not mainly related to the documentation. Those issues include the design of getters (buffers vs pointers) and the functionality of the keyString function. Most of the findings of the analytics-based review were minor issues, that compounded when taking them all together. It uncovered many small pain points and delivered a long list of minor improvements. Those improvements added up and helped improve the usability immensely, particularly through improving the documentation of functions.

Empirical

The empirics-based review was a bit more unique in the kind of issues it uncovered. What set it apart from the other methods was that it showed a lot about the usage of the different functions. This is something that is harder to evaluate with the other two methods, since those methods have no data on how developers use their API. This made it possible to evaluate the usefulness of each function to developers. It helped particularly with cutting down on functions that are almost never used. It also showed functions which can be replaced or unified with other functions. The contrary was also true, it showed which functions Elektra should add to the existing ones, albeit much less than remove. It gave a good sense of which operations are still a bit complicated to achieve with the current API. Having the point of view from actual users therefore provided many interesting insights into the usability of the library. This is a dimension that the other two review methods only could not cover as well as the empirics-based review. The issues found were quantitatively less than the analytics-based review, but they were mostly different kinds of issues. It helped streamline and tidy up the existing API, while also eliminating some paint points through the introduction of new convenience functions.

Reengineering

The reengineering-based method revealed many in-depth questions about the basic functionality of Elektra. It forced the reviewer to consider every design choice made by Elektra and think them through again. This process enabled the reviewer to thoroughly evaluate the existing API. This led to quantitatively little issues compared to the analytics-based review, nut the issues that this review method uncovered prompted interesting insights into how Elektra functions at a core level. It sort of acted as a hybrid between the analytics-based review and the empirics-based review. Because the developer has to revisit every function and their documentation in order to reengineer the API, it forces the reviewer to look at many things that were also covered by the analytics-based review. But because the API had to be reengineered during the review process, the reviewer had to implement and use the whole Elektra API. This leads to in-depth insights into the usability of the Elektra API. Those insights uncovered several issues that concern fundamental aspects of the Elektra API: memory management, Key locking and the handling of binary and string values. It also shone a new light on the insights of the analytics-based review regarding the design of getter functions. Additionally, it showed problems with relying on language-specific behavior such as variadic arguments or union types. While the amount of minor issues was little, partly because many of them already got uncovered by the other two reviews, the review helped reconceptualize many elementary parts of the Elektra API. This led to a draft of the new API that looks considerably different, much more so than the other two review methods.

4.1.2 Strengths of the reengineering-based approach

Questioning the fundamental design of an APi

While the analytics-based and empirics-based review methods found many issues with the existing API and discovered many possible improvements, the reengineering-based method posed fundamental questions about the design of the API in a way that the other two review methods didn't. In particular the empirical-based, but also the analytics-based method, were more concerned with how the existing API could be improved - but they did not lead to fundamental changes in the design of the API. The reengineering-based review method on the other hand prompted fundamental questions about the design of the API and was more concerned with how a usable API could look like rather than with how the existing API could be improved upon. The reengineering process helped take a step back from the current design and the review became more about how the existing API could be transformed into a more usable one, rather than thinking about smaller improvements.

This is evident in the results from the case study. Looking at the suggestions from the reengineering-based review, all of them question fundamental decisions that were part of the API since its inception. Almost all issues found by the reengineering-based method concerned design decisions and fundamental principles of the API. Key locking, variadic arguments and the usage of primitive types were all part of the API since its inception. Only by reengineering the API and re-thinking how it works the design of those parts of the API have been reconsidered. The reengineerd API had different approaches for solving those problems. This is the strength of the reengineering-based approach: It forces the reviewer to take a step back from the problem and helps not to miss the forest for the trees. The other two review methods did a lot less to encourage the reviewer to take a step back and reevaluate fundamental parts of the API, while this process came very naturally with the reengineering-based review method.

This can also be seen in the changes to the API that followed the reviews. While the changes resulting from the first two review methods improved the usability of some functions, the API stayed mostly the same in its fundamental design. Only the reengineering-based review led to fundamental changes of the API and, in turn, a complete overhaul of the general feeling of the API. Many parts of the API were overhauled from the ground up and the resulting API looks and feels very different from the initial API. In order to illustrate this, I want to show a comparison of the initial state of a submodule of the API with the state of the API after the analytics-based review and the state of the API after the reengineering-based review.

An example for the evolution of the API

The initial state of the KeyValue submodule contained a multitude of methods, some of them with very similar purpose. The keyValue() and the keyString() methods could be used to get a reference to the raw value or a string version of the value respectively. It required the caller to know whether a stored value was a string or a binary value. Additionally, the keyGetString() and keyGetBinary() functions provided a way to copy the existing values into a pre-allocated buffer. It required the caller to know all the different functions and their uses. Many of those functions required the caller to additionally pass size parameters, that had to be obtained from other functions that were part of the public API. Those factors made for a very cluttered and seemingly complicated API:

```
const void *keyValue(const Key *key);
ssize_t keyGetValueSize(const Key *key);
const char *keyString(const Key *key);
ssize_t keyGetString(const Key *key, char *returnedString, size_t maxSize);
ssize_t keySetString(Key *key, const char *newString);
ssize_t keyGetBinary(const Key *key, void *returnedBinary, size_t maxSize);
ssize_t keySetBinary(Key *key, const void *newBinary, size_t dataSize);
```

Listing 4.1: Functions contained in the KeyValue module before any API review

The analytics based review found that some of those methods had overlapping functionality. The keyGetBinary() and keyGetString() functions in particular were superfluous and could be removed without any replacements. The tools of the C standard library alone were adequate replacements, since they can be used to copy the value of a variable. The state of the API after the analytics-based review looked like this after the analytics-based review:

```
const void *keyValue(const Key *key);
ssize_t keyGetValueSize(const Key *key);
const char *keyString(const Key *key);
ssize_t keySetString(Key *key, const char *newString);
ssize_t keySetBinary(Key *key, const void *newBinary, size_t dataSize);
```

Listing 4.2: State of the KeyValue module after the analytics-based review

The reengineering-based method suggested abandoning the split of string and binary values altogether. Additionally, it proposed using structs/aliases for a Value rather than resorting to using primitive values throughout the API. This led to a massive simplification of the API. A proposal for how the new API for this submodule could look like after the reengineering-based review:

```
ElektraValue elektraValueNew (const void* value, const size_t size);
ElektraValue elektraEntryGetValue (const ElektraEntry * key);
```

```c
ElektraReturnCode elektraEntrySetValue (ElektraEntry * key, ElektraValue
    value);
```

Listing 4.3: Proposed API of the KeyValue module after the reimplentation-based review [Ini23b]

This is very similar to the API that has been implemented in Rust during the reengineering-based review:

```rust
pub fn value(&self) -> Option<ElektraValue>;
pub fn set_value(&mut self, value: ElektraValue) -> Result<(), ElektraError>;
```

Listing 4.4: KeyValue API from the reengineered API in Rust

This example shows how the reengineering-based method can be used to evaluate the API on a very fundamental level. Rather than improving the pre-existing API, it helps with reconceptualizing the whole API and finding different solutions to the problems the API tries to solve. This is the strength of the review method. It can lead to a radically different API, which would not even have been considered without the reengineering-based review. The reengineering-based review method naturally leads to such fundamental questions in a way that the other two review methods don't.

The API proposed after the reengineering-based revies contains a lot less functions, which in turn improves the usability immensely. In the initial state, the caller had to know a lot about the inner workings of Elektra and then pick the suitable function, which can be overwhelming for beginners. In the final version after the reengineering-based review there is exactly one function with a descriptive name that the user has to call if he wants to retrieve a value. The usage of structs in the parameters and return values also makes the signature of the functions a lot more self-explanatory.

The appendix contains the state of the Elektra public API after every review method (assuming all proposed changes get incorporated). To get a better overview of how the reviews affected the state of the API, please consult the Appendix B for more information. It shows the stark difference of the API before and after the reengineering-based review method and helps show just how much this method helped fundamentally reconsider many parts of the Elektra API.

Providing solutions for the uncovered problems

Having a proof-of-concept at hand, rather than only suggestions for improvements that could be made to the API helped the redesign process of the API immensely. When developers can see the proposed API in use and how it functions, then it is a lot easier to convince others of the benefits of the newly proposed API. With the other methods such proof-of-concepts for a redesigned API can also be created, but the reengineering-based method delivers such proof-of-concept inherently. The other two review methods were good at finding pain points in the design of the API, but they did not offer any immediate solutions. To solve the issues found be the other review methods the developers of

Elektra had to think about alternate solutions and how they could be implemented. The reengineering-based approach on the other hand often showed an alternative solution and a practical example of how it could be used was immediately available. This significantly sped up the process of fixing these issues, since the review method in itself already provided the solutions needed for fixing the usability issues. It also provided a playground for experimenting with alternative approaches before settling on one specific approach. This helped comparing and weighing alternate implementations against each other. In the following reengineering of the API this led to significant savings in time spent coming up with solutions to the usability issues.

Choosing a language for reengineering the API

Having to reengineer the core API in Rust and then exposing a C-FFI compatible interface helped immensely with the review and yielded a lot more results than merely implementing the pre-existing API in Rust. By doing it this way, the whole API had been redesigned from the ground up rather than trying to mimic the existing API. If the reengineering process only used the existing API as a skeleton and then tried to reengineer those methods, then I think a lot less issues would have been uncovered. The results would have been a lot more similar to the other two review methods, which were also more concerned with the existing API and how it could be improved. The act of being forced to redesign the API and then having to reengineer the existing API made the review a lot more thorough. On one hand it served as a validation of the reengineered API, since it was a lot harder to miss existing features and feature parity came very naturally through this process. On the other hand, it showed alternate solutions to how certain functionality can be exposed to the developer in a more user-friendly way.

The redesign in Rust diverged at some points considerably from the design in C. Of course this can also be attributed to fundamental differences in the design of the programming languages. But Rust and C are not too dissimilar in nature to just chalk this up to differences in programming languages. What made Rust particularly suitable is the absence of many constructs known from object-oriented languages. This made a comparison between the API designs more realistic and feasible than if the reengineering language were C++ for instance. Using the same language or a similar language for performing this review method seems to be very important for the effectiveness of the review method.

Following this line of argumentation, it would have made most sense to just reengineer the API using the same language. But this also has some drawbacks that became evident throughout the reengineering process. The design of an API is often governed by the design of a programming language, having to implement the API in a different context naturally leads to thinking differently about functionality. It provides a different viewpoint of the same problem. This can then lead to thinking about how to implement the solutions in a different programming language.

4.2 Threats to validity

In this section I want to give a short overview of possible threats to the validity of my case study. Since the case study consists of multiple different reviews, I will be talking about the possible threats to validity of each study before talking about possible issues with the general results of the case study.

4.2.1 Analytic

One issue with the analytic-based review method is the low amount of people involved in the review. Not every review session has been performed as a peer review. The functions that were peer reviewed only were reviewed by one person and me, which can lead to some issues not being discovered that would have otherwise been discovered if more people had been involved in the review process. All other functions were reviewed by me alone. While the process this method was based on [MMD16] also initially only includes one reviewer, there is always at least a second 'shadow reviewer' involved, which was not the case for most of the reviews performed.

4.2.2 Empiric

KConfig, one of the storage backend implementations, did not use the Elektra core API directly, but rather a thin wrapper of the Elektra Core API in a different programming language which can lead to inaccurate results since the API has only been used indirectly.

The review process I used for this review is usually used to evaluate the quality of the reviewed source code. In my review I reviewed the usage of API that has been utilized to write the code, which can make a significant difference. This also led me to create my own, custom, checklist rather than rely on checklists that have already been tried and tested, such as in the analytics-based API reviews.

The reviewed source code did only include about a quarter of the functions contained in the Elektra core API. This can be a result in itself, that many of the functions are rarely used or needed for utilizing Elektras functionality (as discussed in the section 3.4.2). Nevertheless this is also a potential source of incomplete results, since the review effectively only covered a subset of functions contained in the Elektra API. Possible issues with functions that are not covered would be hard to uncover that way.

4.2.3 Reengineering

Since I already reviewed the API with two different review methods prior to performing the API review, I was already a lot more familiar with the API during the reimplementation-based review. This could lead to the reengineering-based approach yielding more results, since as the reviewer I already had plenty of time to think about the design of the API outside of the reengineering process. In practice, this is not so much of a problem in my opinion, since usually the person performing the reengineering will already be quite familiar with the API.

4.2.4 Case Study

All the reasons from the sections above taken together are possible threats to validity of the case study as a whole. Since I am comparing review methods against each other, the result heavily depends on performing the review methods properly. If one review method has problems and bad results because of this, then the whole review method will get evaluated worse than it actually is. This can skew the results of the comparison.

Due to performing every review method after another, it is impossible for review methods that follow after another review method to uncover issues that have already been discovered by another review method. This can lead to skewing the results towards the earlier review methods, since there is a bigger possible pool of issues to be discovered.

4.3 Research Questions

4.3.1 Research Question 1

Does the reengineering-based approach find usability issues that have not been found by the other approaches?

Compared to the other two review methods, the reengineering-based method yielded many additional usability issues with the API, that the other two methods did not uncover. Those issues were quantitatively less, but they often exposed deeply-ingrained problems with the library, since it provided a wholistic view of the API. This is the strength of the reengineering-based approach. In the case study it led to a reevaluation of many fundamental design decisions of the Elektra API and subsequently a major overhaul of the whole API.

This is evident from the evolution of the API illustrated in the appendix B. The kind of issues that were uncovered concerned themselves with issues that were very deeply ingrained in the API. The other two review methods found many issues as well, but compared to the reengineering-based review they were not as in-depth. When looking at the state of the API after each review, one can see that the API has by far the most changes after the reengineering-based review. This alone is a good indicator that the reengineering-based method uncovered many usability issues that were not recognized after the first two review. The reason for this is because the reengineering-based review led to questioning the design of the API on a very fundamental level.

Additionally, the reengineering-based review shed new light on one of the issues uncovered by the analytics-based review. While the conclusion from the analytics-based review regarding the design of the getter functions was keeping the functions that return a pointer in favor of the functions accepting a buffer. The reengineering-based review showed some additional issues that came with that decision. As discussed in the Results part, there are several downsides that come with that decision that haven't been explored as thoroughly.

4.3.2 Research Question 2

Does the reengineering-based approach find different types of usability issues compared to the other approaches?

The issues uncovered by the 3 review methods differed significantly, which shows that the review methods complimented each other well. While the analytics-based review uncovered many minor issues, the reengineering-based review in contrast uncovered systemic issues that are rooted in the architecture of the API. It prompted questions about the fundamental design of Elektra's API. Exploring those offered interesting insights for designing Elektra for the 1.0.0 release.

Overall there have been many issues found throughout all three reviews methods. All three yielded their own insights and in general complimented each other very well. This lead to many issues being found, which could not have been achieved by only using one review method. The reengineering-based method provided a new viewpoint on some already found issues and uncovered additional issues. This shows that a reengineering-based review method can be an important addition to the already existing review methods. It also showed that the reengineering-based method could be used as a stand-alone method for reviewing APIs, since it combines the types of insights from both. By reengineering the API the reviewer uses the API from the point of view of the developer, similar to the empirics-based review. Additionally, the reviewer has to revisit the documentation and specification of the API and use them in order to reengineer the API. This is similar to the analytics-based approach.

The analytics-based review did a terrific job of improving the documentation of the Elektra API. Without many of the improvements that got added throughout the analytics-based review, the process of recreating the API got a lot easier. From my assessment, the reengineering-based technique would have uncovered many issues with the documentation as well, had they not yet been uncovered by the analytics-based review. This is due to the developer needing to form a good understanding of how the Elektra API exactly works, which in turn leads to a lot of reading of the existing documentation. Had the documentation been in the same state as in the beginning, a lot more information would have been missing. I think, the reengineering-based method would have caught those issues as well, particularly those related to function contracts.

The empirics-based review gave interesting insights into which functions are the most used by developers and which functions were barely used. This made it possible to reason about which functions could be removed from the public API, as well as which functions could be added for convenience. The reengineering-based method lacked insights in that area. While the reengineering-based method can give good ideas on how to overhaul existing functions, the empirics-based method can give ideas about which functions are necessary for the API and which functions could be removed.

4.3.3 Research Question 3

Are any of the usability issues found by the reengineering-based approach related to language-specific behavior?

Two issues found by the reengineering-based approach were directly related to language-specific behavior: The handling of different types of values (string / binary) was accomplished by using a union type, and the usage of variadic arguments in two methods: keyNew and ksNew. Both proved to pose problems for the reenginered version of the API.

In the case of variadic arguments, this showed that the current approach using variadic arguments poses problems in C as well. One of my first experiences using the Elektra library consisted of creating a new Key with the keyNew functions. I did a slight mistake when creating the Key by omitting a certain constant, that needs to be included when creating a Key with additional information via keyNew. The program threw an error message, but it was hard to understand for someone new to the library and made finding the underlying mistake quite hard. Only by looking at the example again and comparing it to my version I was able to find the problem with my keyNew function call. This is not the only problem variadic arguments have. Functions with C-style variadic arguments suffer from several issues: They accept multiple types of arguments, which makes them hard to parse correctly. Since the implementation has to make assumptions about the type of arguments passed in, but has no way of enforcing that, not only proper lists of arguments can be passed. This can directly lead to errors. Additionally, those errors only show up at runtime, making them costly to catch and debug compared to compile-time checked parameters. Because anything can be passed in, functions with variadic arguments can also exhibit undefined behavior when passed an invalid set of arguments.

The other language-specific issue uncovered during the reengineering-based review was related to union types. While they are supported in exactly the same way as in C, which makes a compatible implementation relatively straightforward, it prompted me to think about this implementation detail more. This further caused an inquiry into how Elektra handles values with different types. At the time of the review, Elektra supports string values as well as binary values. Those were implemented using a union type in C. This choice of implementation leads to a lot of added complexity along the way: There had to be different functions defined for string/binary values, the user had to check or know which type is currently stored in the Key, leading to code that is verbose or error-prone. For the implementors it made life harder, since everywhere a value gets used internally, the developer has to carefully consider both possibilities. This in turn makes the exact behavior of functions more complicated, leading to an API that is harder to understand for users.

Additionally, there were some problems discovered with memory management. While memory management is not a thing that is exclusive to C, there are enough differences in how memory management is approached in Rust compared to C. This led to some issues with reengineering the library, especially with regard to providing a library that has a compatible ABI. Those issues prompted further research on how it would be possible to simplify memory management for the user. I found a solution on how to replicate this memory management model in the reengineered API, but this solution still suffered from performance and efficiency problems.

4.3.4 Research Question 4

What are the general up- and downsides of the reengineering-based approach compared to the other approaches?

One upside of the reengineering-based approach is that there is little preparatory work that needs to be done before performing the review. The documents that are used for creating the reengineered API should already exist in the software project in the first place. Those are mainly the documentation and the tests. The documentation was used for getting a definition of the behavior of the functions, while the tests provided a way to verify the correctness of the implementation. Due to Elektra having an extensive API documentation as well as an extensive test suite, the groundwork for the review had already been laid. This enabled me to get started with the reengineering process relatively quickly. If those artifacts do not exist, the reengineering-based approach will naturally get harder but the process can then be used to create proper documentation and tests as well.

The other review methods on the other hand required more preparatory work to be done. The analytics-based review required the reviewer to create a checklist for the review. This checklist was assembled by performing literature research on API design guidelines. From the results of this research, I compiled a checklist applicable to the Elektra API. Only after creating this checklist, the review could be performed. The empirics-based review required me to find suitable projects utilizing Elektra that are good examples for the real-world usage of Elektra. This required careful consideration and thorough research. While this preparatory work wass less compared to the analytics-based review, the reengineering-based review in turn required almost no preparatory work.

While the upfront costs of a reengineering-based review are very low, the review process itself is more complex and time-consuming than the other two review methods. For the analytics-based review, performing the actual peer-reviews also was time-consuming but the nature of the review lends itself well to splitting the effort into manageable chunks that can be scheduled at fixed intervals. This also makes it easy to plan around the review method, since it is quite easy to estimate how long it will take. The empirics-based review and the reengineering-based review do not have a strictly defined end status. The analytics-based method, in this case, just consists of a review of every function and module, then it is considered done.

For the empirics-based review and the reengineering-based review it is harder to give an estimate on how long they will take, since they are more open-ended by nature. The code used in the empirics-based review can be reviewed more than once on several occassions, and almost always new issues will pop up that have not been noticed in prior sessions. It is for the reviewer to decide when enough issues have been found. It is also hard to tell what kind of issues still remain undiscovered by this review method. This is relatively simple for the analytics-based review method, since all the parts that are covered by the checklist are covered by the review. It is easy to tell what factors have been reviewed and which factors have been left out.

For the reengineering-based review, passing all tests of an existing test suite could be used as a factor for determining completeness. Reaching this state is very time-consuming, and many issues can be found before the state of full compatibility with the existing test suite has been reached. This also depends on whether the reengineerd API will actually be used in the future or not. Otherwise passing a majority of the tests rather than 100% of them is often a lot easier to achieve in practice. Also, the goal of the reengineering process is not a perfect reproduction of the API, but rather experimenting with alternative approaches to API design. Performing the review showed, that it is not as complex and time-consuming as creating the API in the first place, since ideally there is already a lot of documentation that can be used as a basis for the reengineering process. This saves a lot of time compared to having to conceptualize the whole API the first time.

Another upside of the reengineering-based review is that the method in its essence is relatively simple and easy to explain. Just reengineer the functionality of the API in another programming language according to the existing documentation. By the nature of the review method the reviewer is forced to look at many different parts of the project: documentation, code, API documentation, function names and parameters. It all comes naturally during the process.

For analytics-based reviews there are a multitude of review methods, each one having different strengths and weaknesses. One needs to weigh their strengths with other review methods depending on the needs of the project and the developers. The empirics-based review suffers from the same problem. There are many different review methods and it is up to the reviewer to choose which one he thinks fits best for the project and the people performing the review. This requires considerable research into different techniques, that is not necessary with the reengineering-based method.

After performing an API usability review and finding issues that need to be addressed, there needs to be a reengineering process for transforming the API according to the results of the reviews. With the analytics-based and empirics-based review the effort for this is often not considered. The reengineering-based approach already includes this reengineering step partly by causing the reviewer to reengineer the whole API. The resulting design and code can then be used to improve the existing API. This lowers the effort spent in the following reengineering process significantly. When considering the effort spent on reengineering the API after a review, one of the disadvantages of the reengineering-based approach does not seem so significant anymore. This is because part of the reengineering effort has already been spent in the reengineering process.

CHAPTER 5

Summary

The reengineering-based review method proved an effective method for reviewing the usability of an API. In the case study it revealed numerous systemic issues with the architecture of Elektra and provided valuable suggestions for redesigning the Elektra API. The reengineering-based approach provided an in-depth reevaluation of many parts of the API of Elektra and proved a very good tool for reengineering an API from the ground up. As can be seen in the state of the API after each review B, the reengineering-based approach led to a complete overhaul of the API, with the result being a leaner and easier to use API when compared with the initial state of the API.

The result of the case study also shows that the reengineering-based method is a good addition to the already existing review methods. All 3 review methods yielded issues with the usability of the API from different point of views, each of them providing unique insights. Many issues uncovered during this process, would not have been discovered if only one review method was used. Depending on the goals of the design review, this should be kept in mind by developers when choosing a review method for their API. Depending on which kinds of issues should be found and fixed, the choice of review method can have a big impact on the type of issues that are discovered.

Compared to the other two review methods the proposed changes were more drastic. This is due the fact that reengineering leads to the reviewer exploring alternate solutions to designing the API, which can then be tested in practice. The other two review methods also showed many issues with the usability of the API, but fell a bit short when it came to finding alternate solutions to problems. They often times uncovered issues, but did not deliver any alternative solutions. Those often had to be found in a subsequent process. The reengineering-based review combined those two things. The reengineering process itself was a good playground for testing different approaches to the design of the API. The finished product also could then be used as a blueprint for the reengineering of the existing API. This proved to be a major strength of the reengineering-based review method when compared to the emprics-based and the analytics-based review methods.

The reengineering-based review proved to be a good hybrid form of the analytics-based and empirics-based review method. This helped in reviewing the API from more than one point of view. While being more time-consuming than the other two review methods, it really forced the reviewer to perform an in-depth review of the API. This makes the invested effort well worth it, since the improvements made to the Elektra API as a result of the review were considerable. The reengineering-based review does not perfectly fit the other two categories, which makes it hard to classify in the scheme proposed by Rauf et al. It combines many aspects of both review methods, which makes it a more well-rounded and complete review method.

While the usefulness and practicability can vary from project to project, in the case of Elektra it proved an immense success. This is also due to the fact that Elektra API, while powerful, is relatively small: The core API contains less than 100 functions. Most of them are basic, fundamental operations. This structure lends itself well to reengineering, since the amount of functions and their complexity stayed manageable. This might have been different, had different parts of the Elektra API been included such as KDB. The cost of labor certainly should not be underestimated and has to be judged individually for each different project.

APPENDIX A

Analytics-based review Artifacts

A.1 Checklist template

```
# {name}

- start = 2021-01-23 18:10
- end = 2021-01-23 18:10
- moderator = Stefan Hanreich <stefanhani@gmail.com>

## Signature

`{signature}`

## Checklist

#### Doxygen

(bullet points are in order of appearance)

- [ ] First line explains briefly what the function does
- [ ] Simple example or snippet how to use the function
- [ ] Longer description of function containing common use cases
- [ ] Description of functions reads nicely
- [ ] `@pre`
- [ ] `@post`
- [ ] `@invariant`
- [ ] `@param` for every parameter
- [ ] `@return` / `@retval`
- [ ] `@since`
- [ ] `@ingroup`
- [ ] `@see`

### Naming

```

```
- [ ] Abbreviations used in function names must be defined in the
    [Glossary](/doc/help/elektra-glossary.md)
- [ ] Function names should neither be too long, nor too short
- [ ] Function name should be clear and unambiguous
- [ ] Abbreviations used in parameter names must be defined in the
    [Glossary](/doc/help/elektra-glossary.md)
- [ ] Parameter names should neither be too long, nor too short
- [ ] Parameter names should be clear and unambiguous

### Parameter & Return Types

- [ ] Function parameters should use enum types instead of boolean types
      wherever sensible
- [ ] Wherever possible, function parameters should be `const`
- [ ] Wherever possible, return types should be `const`
- [ ] Functions should have the least amount of parameters feasible

### Structural Clarity

- [ ] Functions should do exactly one thing
- [ ] Function name has the appropriate prefix
- [ ] Order of signatures in kdb.h.in is the same as Doxygen
- [ ] No functions with similar purpose exist

### Memory Management

- [ ] Memory Management should be handled by the function wherever possible

### Extensibility

- [ ] Function is easily extensible, e.g., with flags
- [ ] Documentation does not impose limits, that would hinder further
    extensions

### Tests

- [ ] Function code is fully covered by tests
- [ ] All possible error states are covered by tests
- [ ] All possible enum values are covered by tests
- [ ] No inconsistencies between tests and documentation

## Summary

## Other Issues discovered (unrelated to function)
```

A.2 Example for a completed review

```
    # keyString

- start = 2021-02-26 16:05
- end = 2021-02-26 16:40
- moderator = Stefan Hanreich <stefanhani@gmail.com>
- reviewer = Robert Sowula

## Signature

`const char *keyString(const Key *key)`

## Checklist

#### Doxygen

(bullet points are in order of appearance)

- [ ] First line explains briefly what the function does
        - [ ] Get the pointer to the string representing the Key's value.
- [ ] Simple example or snippet how to use the function
        - [ ] add example
- [ ] Longer description of function containing common use cases
        - [ ] add explanation about pointers
        - [ ] add explanation about modifications from the user
- [ ] Description of functions reads nicely
        - [ ] `(null)` -> `"(null)"`
        - [ ] `(binary)` -> `"(binary)"`
- [ ] `@pre`
        - [x] add?
- [ ] `@post`
        - [x] add?
- [ ] `@invariant`
        - [x] add?
- [ ] `@param` for every parameter
        - [ ] move before return
        - [ ] key: key -> Key
- [ ] `@return` / `@retval`
        - [ ] pointer to the string representing the Key's value
        - [ ] `(null)` -> `"(null)"`
        - [ ] `(binary)` -> `"(binary)"`
- [ ] `@since`
        - [ ] add `1.0.0`
- [x] `@ingroup`
- [ ] `@see`
        - [ ] `keyGetString()`
        - [ ] `keyGetBinary()`
        - [ ] `keyValue()`

### Naming

- Abbreviations used in function names must be defined in the
  [Glossary](/doc/help/elektra-glossary.md)
- [x] Function names should neither be too long, nor too short
- [ ] Function name should be clear and unambiguous
        - [ ] `keyString()` vs `keyGetString()`
- Abbreviations used in parameter names must be defined in the
  [Glossary](/doc/help/elektra-glossary.md)
- [x] Parameter names should neither be too long, nor too short
- [x] Parameter names should be clear and unambiguous

### Parameter & Return Types

- Function parameters should use enum types instead of boolean types
  wherever sensible
- [x] Wherever possible, function parameters should be `const`
- [x] Wherever possible, return types should be `const`
- [x] Functions should have the least amount of parameters feasible

### Structural Clarity

- [ ] Functions should do exactly one thing
       - [ ] Return Values (null) and (binary)
- [x] Function name has the appropriate prefix
- [ ] Order of signatures in kdb.h.in is the same as Doxygen
       - [ ] swapped `with keyGetValueSize()`
- [x] No functions with similar purpose exist

### Memory Management

- [x] Memory Management should be handled by the function wherever possible

### Extensibility

- [x] Function is easily extensible, e.g., with flags
- [ ] Documentation does not impose limits, that would hinder further
    extensions
       - "(null)" & "(binary)"

### Tests

- [ ] Function code is fully covered by tests
       - [x] Line 198
       - [x] Line 203
- All possible error states are covered by tests
- All possible enum values are covered by tests
- [x] No inconsistencies between tests and documentation

## Summary

Think about changing the return values for 1.0.0. It seems like a hard step,
but also inevitable to do at some point. Now would probably be better than in
the future.

## Other Issues discovered (unrelated to function)
```

APPENDIX B

Proposed state of the API

B.1 Initial

```
Key *keyNew(const char *keyname, ...);
Key *keyVNew(const char *keyname, va_list ap);

Key *keyDup(const Key *source);
int keyCopy(Key *dest, const Key *source);

int keyClear(Key *key);
int keyDel(Key *key);

ssize_t keyIncRef(Key *key);
ssize_t keyDecRef(Key *key);
ssize_t keyGetRef(const Key *key);

int keyRewindMeta(Key *key);
const Key *keyNextMeta(Key *key);
const Key *keyCurrentMeta(const Key *key);

int keyCopyMeta(Key *dest, const Key *source, const char *metaName);
int keyCopyAllMeta(Key *dest, const Key *source);

const Key *keyGetMeta(const Key *key, const char* metaName);
ssize_t    keySetMeta(Key *key, const char* metaName, const char *
    newMetaString);
KeySet * keyMeta (Key * key);

int keyNeedSync(const Key *key);

int keyIsBelow(const Key *key, const Key *check);
int keyIsBelowOrSame(const Key *key, const Key *check);
int keyIsDirectlyBelow(const Key *key, const Key *check);

int keyIsBinary(const Key *key);
int keyIsString(const Key *key);

const char *keyName(const Key *key);
ssize_t keyGetNameSize(const Key *key);

ssize_t keySetName(Key *key, const char *newname);
ssize_t keyAddName(Key *key, const char *addName);

const void *keyUnescapedName(const Key *key);
ssize_t keyGetUnescapedNameSize(const Key *key);

const char *keyBaseName(const Key *key);
ssize_t keyGetBaseNameSize(const Key *key);

ssize_t keySetBaseName(Key *key,const char *baseName);
ssize_t keyAddBaseName(Key *key,const char *baseName);

elektraNamespace keyGetNamespace(Key const* key);
ssize_t keySetNamespace(Key * key, elektraNamespace ns);

const void *keyValue(const Key *key);
ssize_t keyGetValueSize(const Key *key);

const char *keyString(const Key *key);
ssize_t keySetBinary(Key *key, const void *newBinary, size_t dataSize);

int keyLock (Key * key, elektraLockFlags what);
int keyIsLocked (const Key * key, elektraLockFlags what);

KeySet *ksNew(size_t alloc, ...) ELEKTRA_SENTINEL;
KeySet *ksVNew(size_t alloc, va_list ap);

KeySet *ksDup(const KeySet * source);
int ksCopy(KeySet *dest, const KeySet *source);

int ksClear(KeySet *ks);
int ksDel(KeySet *ks);

int ksNeedSync(const KeySet *ks);

ssize_t ksGetSize(const KeySet *ks);

ssize_t ksAppendKey(KeySet *ks, Key *toAppend);
ssize_t ksAppend(KeySet *ks, const KeySet *toAppend);
KeySet *ksCut(KeySet *ks, const Key *cutpoint);

Key *ksPop(KeySet *ks);

int ksRewind(KeySet *ks);
Key *ksNext(KeySet *ks);
Key *ksCurrent(const KeySet *ks);

Key *ksHead(const KeySet *ks);
Key *ksTail(const KeySet *ks);

elektraCursor ksGetCursor(const KeySet *ks);
int ksSetCursor(KeySet *ks, elektraCursor cursor);
Key *ksAtCursor(KeySet *ks, elektraCursor cursor);

Key *ksLookup(KeySet *ks, Key *k, elektraLookupFlags options);
Key *ksLookupByName(KeySet *ks, const char *name, elektraLookupFlags options)
    ;
```

B.2 After the analytics-based review

```
Key *keyNew(const char *keyname, ...);
Key *keyVNew(const char *keyname, va_list ap);

Key *keyDup(const Key *source);
int keyCopy(Key *dest, const Key *source);

int keyClear(Key *key);
int keyDel(Key *key);

ssize_t keyIncRef(Key *key);
ssize_t keyDecRef(Key *key);
ssize_t keyGetRef(const Key *key);

int keyRewindMeta(Key *key);
const Key *keyNextMeta(Key *key);
const Key *keyCurrentMeta(const Key *key);

int keyCopyMeta(Key *dest, const Key *source, const char *metaName);
int keyCopyAllMeta(Key *dest, const Key *source);

const Key *keyGetMeta(const Key *key, const char* metaName);
ssize_t    keySetMeta(Key *key, const char* metaName, const char *
    newMetaString);
KeySet * keyMeta (Key * key);

int keyNeedSync(const Key *key);

int keyIsBelow(const Key *key, const Key *check);
int keyIsBelowOrSame(const Key *key, const Key *check);
int keyIsDirectlyBelow(const Key *key, const Key *check);

int keyIsBinary(const Key *key);
int keyIsString(const Key *key);

const char *keyName(const Key *key);
ssize_t keyGetNameSize(const Key *key);
ssize_t keyGetName(const Key *key, char *returnedName, size_t maxSize);

ssize_t keySetName(Key *key, const char *newname);
ssize_t keyAddName(Key *key, const char *addName);

const void *keyUnescapedName(const Key *key);
ssize_t keyGetUnescapedNameSize(const Key *key);

const char *keyBaseName(const Key *key);
ssize_t keyGetBaseNameSize(const Key *key);
ssize_t keyGetBaseName(const Key *key, char *returned, size_t maxSize);

ssize_t keySetBaseName(Key *key,const char *baseName);
ssize_t keyAddBaseName(Key *key,const char *baseName);

```

```
elektraNamespace keyGetNamespace(Key const* key);
ssize_t keySetNamespace(Key * key, elektraNamespace ns);

const void *keyValue(const Key *key);
ssize_t keyGetValueSize(const Key *key);

const char *keyString(const Key *key);
ssize_t keyGetString(const Key *key, char *returnedString, size_t maxSize);
ssize_t keySetString(Key *key, const char *newString);

ssize_t keyGetBinary(const Key *key, void *returnedBinary, size_t maxSize);
ssize_t keySetBinary(Key *key, const void *newBinary, size_t dataSize);

int keyLock (Key * key, elektraLockFlags what);
int keyIsLocked (const Key * key, elektraLockFlags what);

KeySet *ksNew(size_t alloc, ...) ELEKTRA_SENTINEL;
KeySet *ksVNew(size_t alloc, va_list ap);

KeySet *ksDup(const KeySet * source);
int ksCopy(KeySet *dest, const KeySet *source);

int ksClear(KeySet *ks);
int ksDel(KeySet *ks);

int ksNeedSync(const KeySet *ks);

ssize_t ksGetSize(const KeySet *ks);

ssize_t ksAppend(KeySet *ks, Key *toAppend);
ssize_t ksMerge(KeySet *ks, const KeySet *toAppend);
KeySet *ksCut(KeySet *ks, const Key *cutpoint);

Key *ksPop(KeySet *ks);

int ksRewind(KeySet *ks);
Key *ksNext(KeySet *ks);
Key *ksCurrent(const KeySet *ks);

Key *ksHead(const KeySet *ks);
Key *ksTail(const KeySet *ks);

elektraCursor ksGetCursor(const KeySet *ks);
int ksSetCursor(KeySet *ks, elektraCursor cursor);
Key *ksAtCursor(KeySet *ks, elektraCursor cursor);

Key *ksLookup(KeySet *ks, Key *k, elektraLookupFlags options);
Key *ksLookupByName(KeySet *ks, const char *name, elektraLookupFlags options)
    ;
```

B.3 After the empirics-based review

```
Key *keyNew(const char *keyname, ...);
Key *keyVNew(const char *keyname, va_list ap);

Key *keyDup(const Key *source);
int keyCopy(Key *dest, const Key *source);

int keyClear(Key *key);
int keyDel(Key *key);

ssize_t keyIncRef(Key *key);
ssize_t keyDecRef(Key *key);
ssize_t keyGetRef(const Key *key);

int keyCopyMeta(Key *dest, const Key *source, const char *metaName);
int keyCopyAllMeta(Key *dest, const Key *source);

const Key *keyGetMeta(const Key *key, const char* metaName);
ssize_t    keySetMeta(Key *key, const char* metaName, const char *
    newMetaString);
KeySet * keyMeta (Key * key);

int keyNeedSync(const Key *key);

int keyIsBelow(const Key *key, const Key *check);
int keyIsBelowOrSame(const Key *key, const Key *check);
int keyIsDirectlyBelow(const Key *key, const Key *check);

int keyIsBinary(const Key *key);
int keyIsString(const Key *key);

const char *keyName(const Key *key);
ssize_t keyGetNameSize(const Key *key);
ssize_t keyGetName(const Key *key, char *returnedName, size_t maxSize);

ssize_t keySetName(Key *key, const char *newname);
ssize_t keyAddName(Key *key, const char *addName);

const char *keyBaseName(const Key *key);
ssize_t keyGetBaseNameSize(const Key *key);
ssize_t keyGetBaseName(const Key *key, char *returned, size_t maxSize);

const char * keyGetNextNamePart (Key * k, const char * currentPart);

ssize_t keySetBaseName(Key *key,const char *baseName);
ssize_t keyAddBaseName(Key *key,const char *baseName);

elektraNamespace keyGetNamespace(Key const* key);
ssize_t keySetNamespace(Key * key, elektraNamespace ns);

const void *keyValue(const Key *key);
ssize_t keyGetValueSize(const Key *key);
```

```

const char *keyString(const Key *key);
ssize_t keyGetString(const Key *key, char *returnedString, size_t maxSize);
ssize_t keySetString(Key *key, const char *newString);

ssize_t keyGetBinary(const Key *key, void *returnedBinary, size_t maxSize);
ssize_t keySetBinary(Key *key, const void *newBinary, size_t dataSize);

int keyLock (Key * key, elektraLockFlags what);
int keyIsLocked (const Key * key, elektraLockFlags what);

KeySet *ksNew(size_t alloc, ...) ELEKTRA_SENTINEL;
KeySet *ksVNew(size_t alloc, va_list ap);

KeySet *ksDup(const KeySet * source);
int ksCopy(KeySet *dest, const KeySet *source);

int ksClear(KeySet *ks);
int ksDel(KeySet *ks);

int ksNeedSync(const KeySet *ks);

ssize_t ksGetSize(const KeySet *ks);

ssize_t ksAppend(KeySet *ks, Key *toAppend);
ssize_t ksMerge(KeySet *ks, const KeySet *toAppend);
KeySet *ksCut(KeySet *ks, const Key *cutpoint);

Key *ksPop(KeySet *ks);
Key *ksHead(const KeySet *ks);
Key *ksTail(const KeySet *ks);
Key *ksAtCursor(KeySet *ks, elektraCursor cursor);

Key *ksLookup(KeySet *ks, Key *k, elektraLookupFlags options);
Key *ksLookupByName(KeySet *ks, const char *name, elektraLookupFlags options)
    ;
```

B.4 After the reengineering-based review

This is a proposal of an Elektra developer, Klemens Boeswirth, for a new API for Elektra based on the results of the reengineering-based review, which can be found in the Elektra GitHub repository [Ini23b].

```
ElektraEntry * elektraEntryNew (ElektraNamespace ns, ElektraName name);
ElektraEntry * elektraEntryIncRefCount (const ElektraEntry * key);
ElektraEntry * elektraEntryDecRefCount (const ElektraEntry * key);
void elektraEntryDel (const ElektraEntry * key);

ElektraNamespace elektraEntryGetNamespace (const ElektraEntry * key);
void elektraEntrySetNamespace (const ElektraEntry * key, ElektraNamespace ns)
    ;

ElektraName elektraEntryGetName (const ElektraEntry * key);
ElektraReturnCode elektraEntrySetName (ElektraEntry * key, ElektraName name);
void elektraEntryLockName (ElektraEntry * key);
void elektraEntryUnlockName (ElektraEntry * key);
bool elektraEntryIsNameLocked (const ElektraEntry * key);

ElektraValue elektraEntryGetValue (const ElektraEntry * key);
ElektraReturnCode elektraEntrySetValue (ElektraEntry * key, ElektraValue
    value);

ElektraSet * elektraEntryGetMeta (const ElektraEntry * key);

ElektraReturnCode elektraEntryIsBelow (const ElektraEntry * first, const
    ElektraEntry * second);
ElektraReturnCode elektraEntryCompare (const ElektraEntry * first, const
    ElektraEntry * second);

ElektraEntry * elektraEntryCopy (ElektraEntry * dest, const ElektraEntry *
    src, ElektraEntryCopyFlag flags);
static ElektraEntry * elektraEntryDup (ElektraEntry * key,
    ElektraEntryCopyFlag flags)

ElektraSet * elektraSetNew (size_t prealloc);
ElektraSet * elektraSetIncRefCount (const ElektraSet * ks);
ElektraSet * elektraSetDecRefCount (const ElektraSet * ks);
void elektraSetDel (const ElektraSet * ks);

size_t elektraSetSize (ElektraSet * ks);

ElektraReturnCode elektraSetInsert (ElektraSet * ks, ElektraEntry * key);
ElektraReturnCode elektraSetInsertAll (ElektraSet * ks, ElektraSet * other);

ElektraEntry * elektraSetGet (ElektraSet * ks, size_t index);
ElektraSet * elektraSetGetRange (ElektraSet * ks, size_t start, size_t end);

void elektraSetRemove (ElektraSet * ks, size_t index);
void elektraSetRemoveRange (ElektraSet * ks, size_t start, size_t end);
static void elektraSetClear (ElektraSet * ks)
```

```

size_t elektraSetLookup (ElektraSet * ks, ElektraNamespace ns, ElektraName
    name);
static size_t elektraSetLookupEntry (ElektraSet * ks, const ElektraEntry *
    entry)

size_t elektraSetFindHierarchy (ElektraSet * ks, ElektraNamespace ns,
    ElektraName root, size_t * end);
```

APPENDIX C

Reengineered API in Rust

```
impl KeyName {
    pub fn new (namespace: KeyNamespace, path: RelativePathBuf) -> KeyName;

    pub fn name(&self) -> Option<&str>;
    pub fn base_name(&self) -> Option<&str>;
    pub fn set_base_name(&mut self, base_name: &str);
    pub fn append_name(&mut self, name: &str);

    pub fn namespace(&self) -> KeyNamespace;
    pub fn set_namespace(&mut self, namespace: KeyNamespace);

    pub fn path(&self) -> &RelativePathBuf;

    pub fn is_below_or_same(&self, other: &KeyName) -> bool;
    pub fn is_below(&self, other: &KeyName) -> bool;
    pub fn is_directly_below(&self, other: &KeyName) -> bool;
}

impl FromStr for KeyName {
    fn from_str(name: &str) -> Result<Self, Self::Err>;
}

impl ToString for KeyName {
    fn to_string(&self) -> String;
}

impl KeyLock {
    fn lock(&mut self);
    fn is_locked(&self) -> bool;
    fn unlock(&mut self);
}

impl Key {
    pub fn new(key_name: KeyName) -> Key;

    pub fn name(&self) -> &KeyName;
    pub fn name_mut(&mut self) -> &mut KeyName;
    pub fn set_name(&mut self, name: &str) -> Result<(), KeyError>;
    pub fn set_keyname(&mut self, name: KeyName) -> Result<(), ElektraError>;

    pub fn set_value(&mut self, value: ElektraValue) -> Result<(),
    ElektraError>;
    pub fn reset_value(&mut self) -> Result<(), ElektraError>;
    pub fn value(&self) -> Option<ElektraValue>;

    pub fn meta(&self) -> &KeySet;
    pub fn set_meta(&mut self, meta: KeySet) -> Result<(), ElektraError>;

    pub fn copy_from(&mut self, other: &Key, copy_flags: BitFlags<
    KeyCopyFlags>);

    pub fn lock(&mut self, lock_flags: BitFlags<KeyLockFlags>);
    pub fn unlock(&mut self, lock_flags: BitFlags<KeyLockFlags>);
}

impl FromStr for Key {
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}

impl KeyBuilder {
    pub fn new(key_name: KeyName) -> KeyBuilder;
    pub fn value(mut self, value: ElektraValue) -> KeyBuilder;
    pub fn meta(mut self, meta: KeySet) -> KeyBuilder;
    pub fn build(self) -> Result<Key, KeyError>;
}

impl FromStr for KeyBuilder {
    fn from_str(name: &str) -> Result<Self, Self::Err>;
}

impl KeySet {
    pub fn with_capacity(n: usize) -> Self;

    pub fn len(&self) -> usize;

    pub fn append(&mut self, key: Key);
    pub fn append_all(&mut self, keyset: &KeySet);

    pub fn clear(&mut self);

    pub fn take(&mut self, name: &str) -> Option<Key>;

    pub fn lookup_key(&self, key: &Key) -> Option<&Key>;
    pub fn lookup(&self, name: &str) -> Option<&Key>;

    pub fn get(&self, index: isize) -> Option<&Key>;
    pub fn remove(&mut self, index: isize) -> Option<Key>;

    pub fn values(&self) -> indexmap::set::Iter<Key>;

    pub fn reference_counter(&self) -> u16;
    pub fn increase_reference_counter(&mut self) -> u16;
    pub fn decrease_reference_counter(&mut self) -> u16;
    pub fn set_reference_counter(&mut self, n: u16) -> u16;
}
```

Bibliography

[BBZJ14] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211, 2014.

[BG92] Eric J Byrne and David A Gustafson. A software re-engineering process model. In *1992 Proceedings. The Sixteenth Annual International Computer Software and Applications Conference*, pages 25–26. IEEE Computer Society, 1992.

[Bry99] Bill Brykczynski. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes*, 24(1):82, 1999.

[DR08] Amit Deshpande and Dirk Riehle. The total growth of open source. In *Ifip international conference on open source systems*, pages 197–209. Springer, 2008.

[Ebe09] Christof Ebert. Guest editor's introduction: How open source tools can benefit industry. *IEEE Software*, 26(2):50–51, 2009.

[EHRS14] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering*, pages 632–642, 2014.

[Fag02] Michael Fagan. Design and code inspections to reduce errors in program development. *Software pioneers: contributions to software engineering*, pages 575–607, 2002.

[Fou23] GNOME Foundation. Glib source code repository. `https://gitlab.gnome.org/GNOME/glib`, 2023.

[FZ10] Umer Farooq and Dieter Zirkler. Api peer reviews: a method for evaluating usability of application programming interfaces. pages 207–210, 2010.

[GS16] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[GWL+19] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, volume 1, pages 11–20. IEEE, 2019.

[HAW01] H Rex Hartson, Terence S Andre, and Robert C Williges. Criteria for evaluating usability evaluation methods. *International journal of human-computer interaction*, 13(4):373–410, 2001.

[How06] Michael A Howard. A process for performing security code reviews. *IEEE Security & privacy*, 4(4):74–79, 2006.

[HRA+15] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260. IEEE, 2015.

[Ini23a] Elektra Initiative. Elektra. `https://www.libelektra.org/docgettingstarted/github-main-page`, 2023.

[Ini23b] Elektra Initiative. Elektra source code repository. `https://github.com/ElektraInitiative/libelektra`, 2023.

[ISO11] ISO/IEC 25010. ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models, 2011.

[KDE23] KDE. Kconfig source code repository. `https://github.com/KDE/kconfig`, 2023.

[MAN+13] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE software*, 31(2):78–86, 2013.

[Mey92] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[MF07] Lorraine Morgan and Patrick Finnegan. Benefits and drawbacks of open source software: an exploratory study of secondary software firms. In *Open Source Development, Adoption and Innovation: IFIP Working Group 2.13 on Open Source Software, June 11–14, 2007, Limerick, Ireland 3*, pages 307–312. Springer, 2007.

[MHSH+18] Emerson Murphy-Hill, Caitlin Sadowski, Andrew Head, John Daughtry, Andrew Macvean, Ciera Jaspan, and Collin Winter. Discovering api usability problems at scale. In *Proceedings of the 2nd International Workshop on API Usage and Evolution*, pages 14–17, 2018.

[MKAH16] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189, 2016.

[MMD16] Andrew Macvean, Martin Maly, and John Daughtry. Api design reviews at scale. pages 849–858, 2016.

[Mou17] Ana Maria Da Mota Moura. Awareness driven software reengineering. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 550–555. IEEE, 2017.

[MQO18] Manar Majthoub, Mahmoud H Qutqut, and Yousra Odeh. Software re-engineering: An overview. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 266–270. IEEE, 2018.

[MRARMB+18] Eduardo Mosqueira-Rey, David Alonso-Ríos, Vicente Moret-Bonillo, Isaac Fernández-Varela, and Diego Álvarez Estévez. A systematic approach to api usability: Taxonomy-derived criteria and a case study. *Information and Software Technology*, 97:46–63, 2018.

[NTPM13] Sebastian Nanz, Faraz Torshizi, Michela Pedroni, and Bertrand Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. *Information and Software Technology*, 55(7):1304–1315, 2013.

[PFM13] M. Piccioni, C. A. Furia, and B. Meyer. An empirical study of api usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 5–14, 2013.

[RH96] Linda H Rosenberg and Lawrence E Hyatt. Software re-engineering. *Software Assurance Technology Center*, pages 2–3, 1996.

[RK15] Girish Maskeri Rama and Avinash Kak. Some structural measures of api usability. *Software: Practice and Experience*, 45(1):75–110, 2015.

[RTP19] Irum Rauf, Elena Troubitsyna, and Ivan Porres. A systematic mapping study of api usability evaluation methods. *Computer Science Review*, 33:49–68, 2019.

[Scr72] Michael Scriven. *Die Methodologie der Evaluation*. 1972.

[SG12] Diomidis Spinellis and Vaggelis Giannikas. Organizational adoption of open source software. *Journal of Systems and Software*, 85(3):666–682, 2012. Novel approaches in the design and implementation of systems/software architecture.

[SH14] Samuel Spiza and Stefan Hanenberg. Type names without static type checking already improve the usability of apis (as long as the type names are correct) an empirical study. In *Proceedings of the 13th international conference on Modularity*, pages 99–108, 2014.

[Sne91] Harry M Sneed. Economics of software re-engineering. *Journal of Software Maintenance: Research and Practice*, 3(3):163–182, 1991.

[SSC+18] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pages 181–190, 2018.

[TKM03] Ladan Tahvildari, Kostas Kontogiannis, and John Mylopoulos. Quality-driven software re-engineering. *Journal of Systems and Software*, 66(3):225–239, 2003.

[XAT+20] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. Why reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*, 25:755–789, 2020.

[ZER11] Minhaz F. Zibran, Farjana Z. Eishita, and Chanchal K. Roy. Useful, but usable? factors affecting the usability of apis. pages 151–155. IEEE, 2011.