TU WIEN Informatics

Trend Visualization of Quality Metrics in Software Portfolios

A Visual Expert System for Software Quality Analysis on Portfolio Level

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Patric Genfer
Matrikelnummer 01528942

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 12. Oktober 2020

Unterschrift Verfasser

Unterschrift Betreuung

TU WIEN Informatics

Trend Visualization of Quality Metrics in Software Portfolios

A Visual Expert System for Software Quality Analysis on Portfolio Level

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Patric Genfer
Registration Number 01528942

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 12th October, 2020

Signature Author

Signature Advisor

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Trend Visualization of Quality Metrics in Software Portfolios

A Visual Expert System for Software Quality Analysis on Portfolio Level

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Patric Genfer
Matrikelnummer 01528942

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 12. Oktober 2020

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Patric Genfer

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, 12. Oktober 2020

Patric Genfer

Kurzfassung

Softwareportfolios heutiger Unternehmen bestehen aus einer Vielzahl modularer, teilweise äußerst polyglotter Einzelkomponenten. Das Qualitätsmanagement dieser Softwareportfolios erfordert neue Ansätze, da hier die Überwachung von Qualitätsmetriken auf Projektebene allein nicht mehr ausreichend ist, um wichtige strategische Entscheidungen zu treffen. Während Qualitätsmanager bei der Analyse und Wartung einzelner Softwareprojekte auf eine Vielzahl an Untersuchungen und Werkzeugen zurückgreifen können, ist das Qualitätsmanagement auf Portfolioebene, trotz seiner zunehmenden Relevanz, bisher ein wenig beachtetes Gebiet, weder aus akademischer, noch aus kommerzieller Sicht. Um diese Forschungslücke weiter zu schließen, präsentiert diese Arbeit den Prototypen eines Expertensystems zur Visualisierung und Analyse von Softwaremetriken auf Portfolioebene.

Die konkreten Anforderungen an den Prototypen ergeben sich aus den aus verschiedenen Quellen gewonnenen Informationsbedürfnissen von Qualitätsmanagern im Portfoliobereich. Die Implementierung selbst erfolgt in zwei separaten Teilkomponenten: Einem Data-Mining-Prozess, um die erforderlichen Qualitätsmetriken aus einem Softwareportfolio zu extrahieren und aus dem eigentlichen, interaktiven Expertensystem zur Visualisierung der gesammelten Trenddaten.

Eine abschließende, szenariobasierte Expertenevaluierung konnte bestätigen, dass der Prototyp durch seinen holistischen Ansatz und seine Konfigurierbarkeit neue Möglichkeiten im Portfolio Qualitätsmanagement bietet, die so von existierenden Tools bisher nicht angeboten werden. Die durch den Prototypen gewonnen Erkenntnisse, ebenso wie die im Rahmen dieser Arbeit gesammelten konkreten Informationsbedürfnisse von Qualitätsexperten auf Ebene der Software Portfolios, legen einen Grundstein für mögliche weitere Forschungen in diesem Gebiet.

Keywords: *Softwarequalität, Portfolios, Trendanalyse, Softwaremetriken, Datenvisualisierung, Source Code Repository Mining, Qualitätsmanagement*

Abstract

Compared to past monolithic approaches, today's companies host a large amount of various, modular and also very polyglot software components in their portfolios. Ensuring high software quality standards within these portfolios is a huge challenge and requires new strategies, as monitoring quality metrics only on the project level is no longer sufficient enough for making important decisions. Regarding the management of single software repositories, quality managers have access to a large number of research papers and tools, but the same is not true when it comes to quality analysis on portfolio level: Despite its increasing relevance, there does not exist much supportive material, neither academically, nor commercially. In an attempt to close this research gap further, this work presents the prototype of an expert tool for visualizing and analyzing software quality metrics at portfolio level.

The specific requirements for developing the prototype are determined by collecting the concrete information needs of quality managers from various sources. The implementation itself is divided in two separate parts: First, a data mining process, used to extract the necessary quality metrics from a software portfolio, is developed, and based on that, the actual interactive expert system to visualize the collected trend data, is realized.

A concluding scenario-based expert evaluation confirmed that the prototype, due to its holistic approach and various configuration options, provides analyzing and monitoring software portfolios' quality in a way actual tools in this area do not yet provide. Furthermore, the insights gained through the development of this prototype, together with the concrete information needs of quality experts on the portfolio level collected in this work, lay the foundation for further research in this area.

Keywords: *Software Quality, Portfolios, Trend Analysis, Software Metrics, Data Visualization, Source Code Repository Mining, Quality Management*

Contents

CHAPTER 1

Introduction

Software applications have become an essential part of the value chain in many companies, be it through in-house development of custom solutions or application-specific adaptions of existing systems. By using individual applications that are precisely tailored to its individual processes and domains, a business can also gain important advantages over competitors [1]. As a result, today's companies often own a pool of different software solutions, which they have to manage, develop and maintain - a *software portfolio*.

Whereas, a few years ago, managing software portfolios was primarily considered as a matter of larger corporations to organize their few, mainly monolithic business applications, the situation today is much more differentiated: Even small companies own and use a considerable amount of various applications and technologies, and managing these efficiently promises various benefits, like better strategic alignment and resource allocation [4]. Furthermore, current development in this area suggests that this trend might even increase in the future, as the number of applications and projects available within a company might continue to grow. For this, several reasons can be identified: On the one hand, server architectures are moving more and more from single, monolithic blocks in the direction of distributed microservices, where individual services encapsulate different domain or business concepts and work in isolation as much as possible [5]. This approach increases the scalability and maintainability of the applications, but at the same time also increases the total number of applications within the portfolio, especially since different versions of these services can often run in parallel during production for reasons of backward compatibility. Also, and especially in the tech industry, an increasing trend regarding acquisitions can be observed in recent years (see Figure 1.1). Within the framework of these acquisitions, the software solutions of acquired competitors will be merged in the buyer's own portfolio, which might often result in many applications with similar or overlapping functionality coexisting next to each other in the same portfolio.

[1] `https://www.visualcapitalist.com/interactive-major-tech-acquisitions/`, last accessed on 30.03.2020

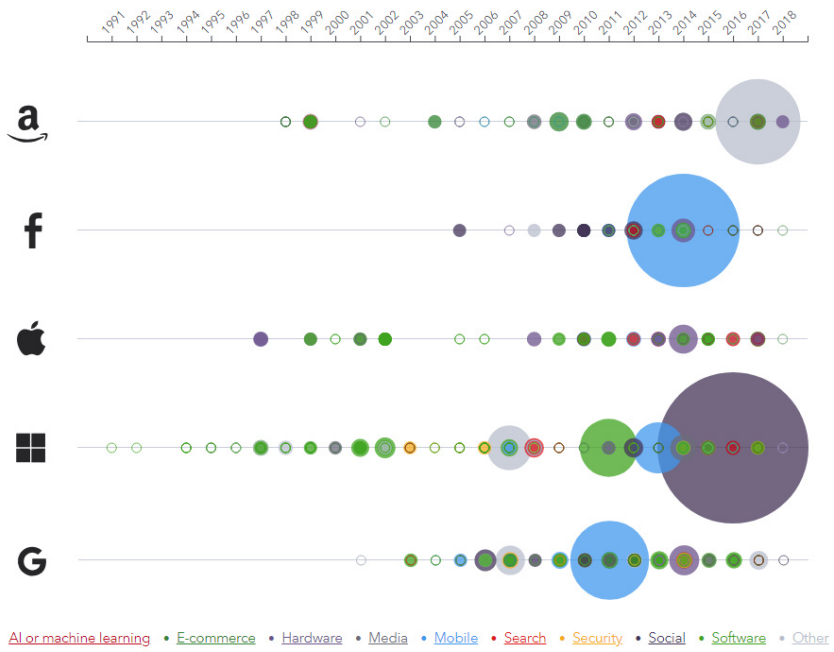


Figure 1.1: Acquisitions by the five largest tech companies since 1991, grouped by business domains. The radius of the circle illustrates the size of the company that was acquired[1].

Another aspect that might affect the growth rate of software portfolios is also the increased use of software product lines [6]. Here, a company uses a common application platform with reusable parts and adapts various aspects of the platform depending on the product that has to be implemented. While this approach can help in reducing the overall size of the code base, implementations of product specific use cases might now coexist next to each other within the same portfolio.

This increasing number of applications within a single portfolio leads to new challenges, especially considering quality related management aspects. In particular the adoption of micro services might lead to an inhomogeneous portfolio, consisting of several small apps, written in various programming languages and using a wide palette of diverse backend technologies [7]. Also, since the various services might implement use cases of different severity levels, diverse quality standards will have to be applied to each of them. Nevertheless, it should be stressed out, that while the actual quality characteristics can be different for each project, the common aspect of monitoring, evaluating - and maybe even comparing - these metrics for the whole portfolio, still remains. The challenges and problems that arise from the quality analysis on this larger scale are worth further consideration.

1.1 Problem Description

When it comes to analyzing the quality of a software system, many different approaches and actions have been introduced during the years, the use of software quality metrics being one of them [8]. Software metrics map specific properties of a system, like its complexity or maintainability, to a corresponding numeric value that indicates how weak or strong this property is in the examined system. While software metrics are a commonly used concept, their use is often controversial, especially in cases where the various metrics, like source lines of code (LOC), are considered only in isolation [9, 10]. Nevertheless, they are a popular tool when it comes to software quality analysis.

Due to the growing number of publicly available source code repositories (GitHub alone currently hosts over 35 million public repositories[2]), the topic of mining and analyzing software metrics has increasingly come into the focus of current research again (for instance, see [11], [12] or [13]), and in addition, a wide range of tools is also available to assist users by gathering software metrics from different repositories [14].

However, much of this effort, both in academical area and by toolmakers, focuses on individual source code repositories in isolation. Cross-project analyses, as they would be required for portfolios, are rarely done, and if they are, the focus lies only on discrete points in time like in [11] and not on historical trends. Even most tools, both commercial and freely available ones, provide only visualization and analysis features that focus on single project analysis, and, in some rare cases where portfolio visualization is supported, the feature set is quite limited (see Section 3.2 for an in-detail comparison of the different tools and their features).

Although the growing number and size of software portfolios suggest that there will be an increasing demand for suitable visualization solutions, some challenges still have to be overcome in this area:

- So far, the concrete information needs of software quality managers regarding software portfolios are not well understood yet. Although extensive studies have already been carried out in the area of single-project quality analysis [15], it has not been finally clarified so far to what extent these results could also be applied to their superior portfolios. Any further investigation in this area must therefore initially aim to gather more insights here.
- Monitoring a complete software portfolio over a longer period of time would result in a dataset much larger than the ones collected during single project analysis. This is particularly problematic because quality managers often have only limited resources available for analysis tasks, and therefore the data should ideally be already highly aggregated [16]. The only possibility to do this in a reasonable amount of time would be through an automated process, which has to be defined and implemented first [17].

[2] `https://github.com/search?q=is:public`, last accessed on 30.03.2020

- Various visualization methods already achieve a fairly high information density when displaying the metrics of individual projects. Scaling these visualizations up for several projects can quickly lead to a possible information overload for the viewer [2]. This makes it difficult to identify relevant pieces of information and therefore complicates the decision making process.

- Since software projects can be very inhomogeneous, even within the same portfolio, there do not exist any general metric thresholds which would be valid for all them. While some portfolio artifacts might be considered as system-critical and therefore require stricter quality thresholds, these restrictions might not apply to all members within the portfolio. A possible visualization solution must take this into account and should be able to provide individual health criteria per project.

- Individual applications might have different development cycles and speed, therefore comparing their metrics over time is not easily possible. Also, other meta-data like number of commits or size and experience of the development team might vary widely within a portfolio.

- Even if the quality analysis starts at the portfolio level, experts must still be able to increase the level of detail to drill down into single projects and artifacts to better understand the overall context within the portfolio [15]. These different levels of detail must be taken into account when designing the expert visualization system.

Finding an adequate solution for these challenges will be the major purpose of the present work.

1.2 Motivation

Managing the quality of their software portfolios can provide meaningful benefits even for smaller companies, as this would allow them to better organize their limited resources and could also improve their decision-making process [4], but this requires all measurements to be based on a solid data foundation [18]. When it comes to single project analysis, quality managers and data experts can already rely on a wide palette of available tools, like SonarQube[3], and research.

Extending the quality management process to include software portfolios as a whole is, however, a problem that has been little investigated so far, but if the interest in software portfolio management continues to grow, this topic might certainly gain traction in the future, even besides pure academic research.

But in order to provide a suitable visualization solution here, the concrete information needs must first be analyzed and a corresponding data mining and visualization prototype must be designed, implemented and evaluated. The following work contributes to this

[3] `https://www.sonarqube.org/`, last accessed on 29.09.2020

research field by providing such a prototype, which supports quality experts during their analysis and helps them complying with their quality guidelines.

1.3 Expected Results

The research problem that is covered by the present work can be described as:

> *How would an expert visualization system have to be designed to support software quality managers in analyzing software portfolios?*

Investigating this topic further, two main tasks which have to be solved can be identified here: (1) The development and evaluation of the expert visualization prototype and (2) a mining tool chain that gathers and processes the relevant quality data for the visualization. Regarding the visualization system, the research problem can be broken down further into the following research questions:

RQ1: What are the information needs of quality managers when analyzing software portfolios, are there any differences compared to quality analysis at project level?

RQ2: Which concrete requirements for an expert visualization system can be derived from these information needs?

RQ3: How can the specific requirements and use cases at project or portfolio level be implemented by the visualization system?

RQ4: Which visual concepts are suitable to visualize the quality metrics on portfolio level without causing information overload?

RQ5: Does the prototype satisfy the information needs of quality experts and what improvements do they propose?

Secondly, an automated data mining tool chain has to be designed and implemented to gather the relevant software portfolio quality metrics. As this tool chain is mainly used as means to collect the data for the later visualization, its development time should be reduced by relying on external tools (like metric calculation providers) as much as possible. Regarding this repository mining process, the following research questions will have to be answered during this work:

RQ6: How must an automatic process be organized to collect large amounts of data from different repositories? Which different steps are necessary for this task, and how do these steps interact with each other?

RQ7: How could other existing applications, like quality metric providers, be integrated into the process, in order to minimize the development effort?

RQ8: What variable extension points would be necessary within the process to support different portfolio-dependent requirements, for instance, to allow integration of different metric- or source code providers?

1.4 Structure

The present work is organized as follows:

- *Chapter 1*, this chapter, explains the motivation behind the work and what contributions it provides to the research field of software portfolio quality visualization and analysis. While there are suitable tools for single project analysis, quality managers are currently faced with a lack of tool- and research support for analyzing software portfolios. Based on this starting position the idea of an expert visualization prototype is presented.
- *Chapter 2* will outline the theoretical foundation on which this work is based. All relevant concepts and terms that are important for this work are briefly presented here. This includes important notions, such as software portfolios or quality management, but also various visualization concepts and some selected software quality metrics.
- An overview of the current state of art is given in *Chapter 3*. It introduces the most relevant research papers on which this work is based, but also points out where this present research can be distinguished from them. Furthermore, a selection of some of the most popular quality analysis tools, together with their capabilities, are briefly introduced.
- *Chapter 4* describes the methodology that was used during this work, presenting both the procedure for literature research as well as the processes for the development and evaluation of the prototype in detail.

- *Chapter 5 and 6* form the main part of the work: The conception and implementation of the prototype system. The system consists of two parts, the mining toolchain and the visualization prototype. Both were developed separately and consecutively and will, accordingly, be discussed in two separate chapters. *Chapter 5* describes the work that was necessary to develop the repository mining tool chain. It starts by analyzing existing research in this area and based on these findings, defines the requirements for the tool chain. It concludes by describing how these requirements were implemented in the scope of this prototype, also pointing out some relevant implementation details, together with an experiment based evaluation of the mining process. *Chapter 5* does the same for the expert visualization prototype, but here the information needs were determined through additional qualitative expert interviews. The chapter concludes with a presentation of the final visualization prototype and how the gathered requirements were realized by its features.

- *Chapter 7* describes the evaluation process of the expert visualization system, based on a scenario based expert evaluation. The chapter introduces the methodology and test plan that was used to carry out the survey, followed by a detailed description of the various test scenarios. The actual test execution and the evaluation of the test results form the conclusion of this chapter.

- *Chapter 8* finally summarizes the results and insights that could be gained during the creation of this work and provides an outlook towards possible future research in this area.

CHAPTER 2

Foundations

The following pages give an overview of the theoretical foundations on which this work is based. The chapter begins with an explanation of the used domain specific terminology and definitions and then moves on to individual subject areas that are also highly relevant in this work.

2.1 Domain Concepts

First of all, it should be clearly defined what is meant by the term software portfolio in the context of this work. According to Elonen and Artto [2] and Simon, Fischbach, and Schoder [3], one can speak of a software portfolio as soon as the individual software projects compete for a common, limited amount of resources and are subject to central administration, which allows an efficient and effective management. While this definition already describes the characteristics of a portfolio comprehensively, one essential aspect is not considered here: The quality management of these portfolios. In order to take this quality aspect into account when evaluating software portfolios in the context of this work, the aforementioned definition should therefore be expanded:

Definition 1 *A software portfolio describes a group of applications that are subject to a common quality standard and process.*

Therefore, whenever the term *software portfolio management* is used in this work, this quality aspect is always implicitly included.

Beyond the term *portfolio*, several other important concepts are relevant in the context of this study, which are therefore briefly presented. Most of these domain concepts were collected by consulting the corresponding literature, like, for instance, Kiefer, Bernstein, and Tappolet [55], where a complete software ontology model was defined which was used here in parts to identify relevant domain entities.

Repository/Project For this prototype, this term refers exclusively to source code repositories. Therefore, a repository represents the entirety of all source files that are part of a software project and also contains all versions of these source files that were created during the whole project life cycle. While, depending on the concrete use case, a distinction can be made between the more abstract term *project*, which describes a component of a portfolio, and *repository*, which is a more technical term that describes the concrete place (local or remote) where the source files are hosted, this distinction was not considered relevant here and therefore was not taken into account any further.

Snapshot A snapshot defines the state of a repository at a defined point in time. Other synonyms like *commit* [54] or *revision* [55] can also often be found in literature. While these terms are often associated with a specific technology, the term *snapshot* is more technology-agnostic and is therefore preferred in this work. Every snapshot contains a set of meta data like the time stamp when it was created, a documentation message that describes which changes were applied to the repository at this snapshot and others.

Metric In general, a metric is a function that describes a specific, measurable characteristic of a a system. In the context of software development, metrics are often used to measure specific qualities of a software artifact, likes its maintainability [8], which makes them an essential part of any quality analysis, be it on single artifact, project or even portfolio level. Since there is a wide variety of metrics available, the final decision which metrics to use will highly depend on the concrete use case and cannot be generalized. This study only uses some exemplary metrics which calculation is provided by an external metric provider. The final selection is described in more detail in Section 2.2.

Measurement/Metric Value A metric value [55] expresses the specific measured value of a metric for a software artifact at a given point in time. These measurements can change over time, as artifacts are subject to change during the development process. Therefore, a metric value for an artifact is only valid for the specific snapshot where it was recorded.

Metric Provider/Collector A metric provider or collector is a system that calculates and provides metric values for the artifacts of a given snapshot of a project. Since implementing these calculations from scratch can be very resource intensive, this study relies on external providers for this task.

Artifact A software artifact defines any atomic element for which a metric value can be calculated [55]. Depending on the type of artifact, not all possible metrics might make sense. For simplicity, this prototype focuses on three different artifact types which are most common among different object-oriented languages, namely *files*, *classes* and *methods*. Other, higher order concepts like packages, namespaces or assemblies are not considered here as they are mostly language specific.

2.2 Metrics

For the implementation and later use of the prototype during this study, an exemplary selection of different metrics was made. It must be noted that this selection is in no way complete or representative, as the concrete selection of quality metrics depends heavily on the individual use cases of the respective quality managers. Due to limited resources, some restrictions had to be made regarding this selection. Since it was not planned to create any executable resources during the mining process, all metrics that require a preliminary compilation could not be calculated. Unfortunately, this also applies in particular to the *test coverage* metric which is generally considered to be a quite important metric by many experts. The following categorized enumeration presents a subset of the finally selected metrics, together with their benefits from a quality manager's point of view. The categorization follows the approach provided by [47] and differentiates the metrics into more traditional ones that are best applied to procedural instructions and metrics which are more related to object-oriented design.

2.2.1 Traditional Metrics (Volume / Complexity)

Source Lines of Code (*SLOC*)

This metric counts the total amount of lines of source code for a specific source code artifact. Values are available for classes, methods, files. By accumulating the lines of all files, the present repository miner also calculates the total amount of lines within a project. While this can be a useful metric for quality managers to evaluate the size of a project, only considering lines of code in isolation can be problematic, as the pure size of an artifact does not always allow drawing conclusions regarding the quality or complexity of this piece of code. Instead, *SLOC* should best be used in conjunction with other metrics [9]. The amount of lines of code can also depend on the used programming language, but since this prototype focuses on Java and C#, the differences should be negligible [43] (see also *M.R6* in this context).

Comment Percentage / Comment to Code Ratio

The ratio of comment to code is strongly related to the previous metric, as it expresses how many lines of code documentation exist compared to the total lines of code [47]. Since a line can contain both code and comment and a file can as well contain much more documentation than instructions, a ratio of more than 1 is possible. Although, as a standalone metric, the comment percentage does not say much about the quality of a system, it might be interesting to investigate whether its trend correlates with other metrics and how this correlation could be interpreted regarding the maintainability and usability of the code.

Cyclomatic Complexity (CC)

This metric measures the possible paths that can be taken through a program or a method [48]. To describe it in a simplified way, this is achieved by interpreting the program (or better - the algorithms that represent the program logic) as a graph, where every single node represents a set of program instructions that can be executed sequentially. Every branch in the program logic can then be viewed as an edge that connects two of these sequential instruction blocks [48].

For analyzing procedural program sequences, the usage of cyclomatic complexity certainly has its justification, but evaluating the complexity in programs based on other paradigms can become difficult: Object-oriented programs use message passing (often in the form of calling instance or class methods) to delegate decision making to other objects, in that way, while the calling code might look less complex, the real complexity could be hidden in the called method. [47].

A different situation can be found for reactive programming paradigms: Here the program's control flow is not defined by sequential instructions, but instead the program reacts on external events like user interaction and further data processing is implemented by stream transformations [49]. While these event-handling and notification mechanisms can become quite complex, their complexity is not always reflected by high CC values.

Regardless of that, however, identifying single methods with a high CC value can still be beneficial, as this could possibly indicate implementations which are more complex than necessary and therefore require further investigation.

2.2.2 Object-Oriented Metrics

Based on Chidamber and Kemerer [22], two main factors for evaluating object-oriented systems are *coupling* and *cohesion*. To better understand these two concepts, their formal descriptions should be briefly introduced here:

Assuming that two objects X and Y can each be described as a tuple

$$X = \langle x, p(x) = \langle \{M_X\} \cup \{I_X\} \rangle \rangle$$

$$Y = \langle y, p(y) = \langle \{M_Y\} \cup \{I_Y\} \rangle \rangle$$

where the first tuple member x or y represent the respective object's instance (the *this* or *self* pointer) and $p(...)$ is a collection containing all methods (M) and instance variables (I) of the corresponding object, then the *coupling* of the two objects X and Y can be described as

1. any access a call to M_Y does on M_X or I_X or
2. any access of M_X on either M_Y or I_Y

Strong coupling can be problematic, as it complicates the reuse of a class and also decreases the modularity of a system. Furthermore, a strongly coupled architecture is harder to maintain, as changes in one class are more likely to affect other classes [47].

Cohesion on the other hand can be expressed by the degree of similarity of an object's instance methods (expressed by the Greek letter σ), whereby the similarity of two methods is defined as the intersection of all instance variables that are used by both methods [22]:

$$\sigma(M_1, M_2) = \langle I_1 \rangle \cap \langle I_2 \rangle$$

The higher the similarity of method M_1 and M_2, the better the cohesion of the object, since both methods access the same set of instance variables and therefore probably also implement related functionality.

There exist different metrics to evaluate both of these concepts, the two most commonly used are:

Coupling between object classes (*CBO*)

This metric measures the coupling between two classes by counting how often a specific class accesses members of another class to perform its tasks. An increased *CBO* value can be an indicator that the system might require a redesign, as there is possibly too much coherent logic that is spread across too many classes. This could also be a sign that a possible domain concept is hidden and split among different classes, but instead should better be modeled explicitly [30].

Lack of Cohesion in Methods (*LCOM*)

Among the different ways of calculating this metric [47], the later used metric provider determines for every instance field, which percentage of methods is accessing this field. The average of these percentages will then be subtracted from 100 to get the final *LCOM* value of a class (expressed in percentage). To illustrate this metric by an example: Assuming a class has many methods and instance variables, but each method accesses only a limited set of these instance variables, then the overall *Lack of Cohesion in Methods* value for this class is very high, as there is only a small percentage of methods accessing each single instance variable. One reason for this could be that the class tries to fulfill several, non-related purposes and should better be split up into different classes, each of it with its own purpose. This concept is also called the *Singe Responsible Principle* [50] and can help improving the maintainability and reusability of a class.

Although the following two metrics are not directly related to coupling or cohesion, they can nevertheless provide interesting insights into the structure of an object-oriented system and therefore represent a useful addition to this enumeration.

Depth of Inheritance Tree / Number of Children

In languages like Java or C#, the static inheritance hierarchy of a class can be interpreted as a tree, starting at the base class and going down one level for every derived class. The

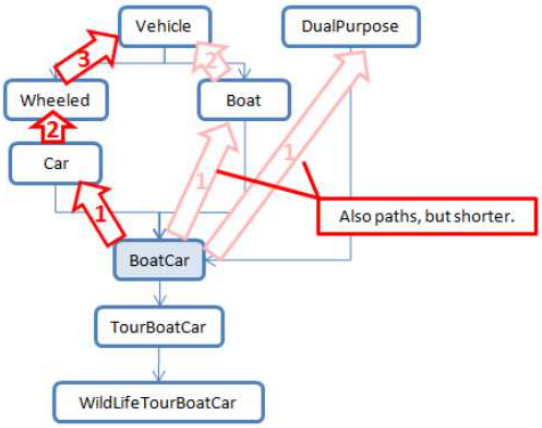


Figure 2.1: The *Depth of Inheritance tree* metric measures the longest path from a given class to the hierarchy's root class[1]

structure of this tree reveals some interesting insights not only about the relations of entities within the modelled domain, but also by how many classes a specific class can be influenced [22]. For example, if a class has an inheritance depth of three (see Figure 2.1), that means that changes in any of its three base classes could possibly affect the class itself, thus making the system harder to maintain.

Per definition, only the maximum distance to a base class is counted [22], but since the languages analyzed by this prototype support only single inheritance, there will always be only one possible inheritance tree (apart from interface implementation, which is not considered here).

The same applies to the *Number of Children* metric, where the number of classes directly derived from a specific class is measured. Changes to this single class could therefore affect all immediate child classes.

High values for these two metrics could indicate a software architecture that has become very complex and difficult to maintain. Very low values on the other hand could point to a system which code reuse capabilities and expressiveness could possibly be improved by defining a more sophisticated class hierarchy.

2.2.3 Thresholds

Defining valid and general accepted thresholds for aforementioned metrics is a difficult undertaking, as most of these metric values might often depend on their context. Furthermore, individual applications can vary widely in their architecture and implementation and therefore comparing their metrics directly is very difficult. While in one case, high

[1] `https://scitools.com/support/metrics_list/?metricGroup=oo`, last accessed on 19.02.2020

complexity might be required to map comprehensive business processes correctly, the same metrics could also indicate a serious problem when calculated for a different application.

There exist some research papers that already focused on finding useful metric thresholds, with somewhat differentiating results. El-Emam [51], for instance, analyzed different studies regarding possible metric thresholds but came to the conclusion that a threshold effect for inheritance depth or cohesion could not always be identified, at least regarding the fault-proneness of a system - other system qualities like maintainability were not part of their investigation.

A different approach was applied by Ferreira et al., where the distributions of metrics from 40 different open source projects were calculated and analyzed [12]. Assuming that the most common values could be interpreted as valid thresholds, they conclude that the ideal inheritance depth should be around two and the ideal lack of cohesion in methods should not be more than 20.

A similar study, but with a larger scope, also analyzed the distribution of metrics among open source projects [11]. Based on their measurements, they were also able to identify some average values for different metrics, that could be used as possible thresholds.

Since actual research seems to partly disagree on possible metric thresholds as these can depend on many different factors (an observation that was also mentioned by some of the experts during the interviews accompanying this research), this prototype implementation does not provide any predefined metric thresholds.

2.3 Information Needs

Software quality managers as stakeholders have very specific information needs regarding the quality analysis of software portfolios. Since identifying these needs forms the foundation for any further requirement specification of the visualization prototype, extensive literature research was carried out to cover the various aspects of these needs. All of the findings that were collected during this process will be introduced in the following section.

What was initially striking during the literature research was the fact that it is relatively difficult to define the role of a software quality manager concretely and to differentiate it from other, similar positions. Often there is no clearly dedicated position for this role. In some companies, the relevant quality tasks are mainly carried out by software developers, with or without a leading rule [15], while in other companies, these responsibilities are assigned to project managers or other decision makers [60]. However, the problem arises from this, that quality analysis is often not the main and sole responsibility of this employee and therefore only limited resources could be provided to solve any tasks related to this area. [17].

According to Card and Jones [60], the need for information about quality analysis must also be a fundamental prerequisite for collecting any data. As long as there is no person

with such a requirement, there is also no need to collect and analyze these data. Based on this conclusion and to not accidentally exclude any possible stakeholders, the role of a software quality manager in the context of this research can be defined as follows:

Definition 2 *A software quality manager is any person with a crucial need of gathering information about the quality state of a software project or portfolio. Furthermore, she or he should use this information to derive decisions that could affect the further development of the analyzed artifacts.*

On the level of software projects, two root causes for these information needs can be identified [60]:

1. The gathered information are necessary to reach the pre-defined project goals and
2. the information can be used to remove any obstacles to achieving these project goals.

Moving one level up on the area of project portfolios, according to Elonen and Artto [2], there are three additional reasons for collecting quality information:

1. Maximizing the portfolio's value
2. Linking the portfolio to the overall companies' business strategy and finally
3. Balance the available resources between the different projects within the portfolio.

Achieving these goals results in different information needs, both on project and on portfolio level. In particular, this matter was already comprehensively examined on project level in a study among 110 Microsoft employees in 2012 [15]. The authors of the study made a distinction between software engineers and managers - although the border between these two had been relatively fuzzy - and found out that managers prefer to fall back on data and metrics when making decisions while software engineers primarily relay on their personal experience. This finding is important because it underlines the necessity of collecting quality metrics over the course of a project cycle. The collected data is mainly used for historical trend and current state analysis and less for predicting future developments. According to the authors, this might be due to the lack of appropriate prediction models [15], but it should be noted at this point, that their survey was carried out a few years ago, and especially prediction models have significantly improved in the last years thanks to new machine learning approaches. However, since there are no recent studies in this regard, the present work will also focus on the visualization of historical and current metric values, as suggested by the authors.

The level of detail of the artifacts, for which quality managers want information, is also interesting: Whereas executives prefer an overall, high-level view of the project,

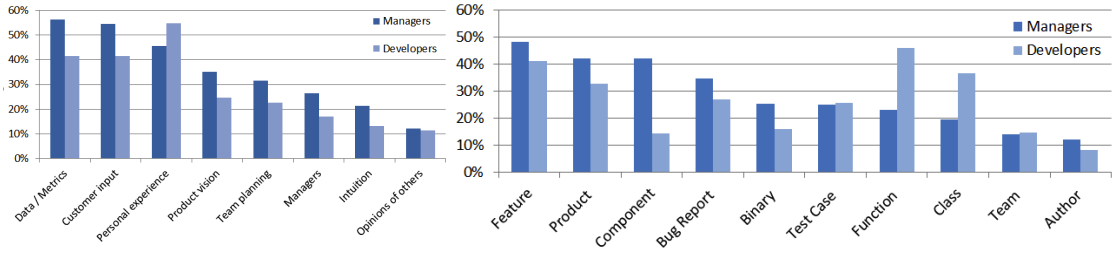


Figure 2.2: Importance of factors (left) and artifacts (right) that managers and developers use for decision making [15]

developers are more interested in individual code artifacts, such as classes and functions. However, this does not necessarily mean that project managers are not interested in these low-level details, rather it results in being able to drill down into specific single artifacts from a high-level view and navigate between these different layers of information. [15]

In summary, it can be said that the information required by managers and developers can be very complex and manifold, and there is both interest in customer-specific data like bug and crash reports, but also in quality related software metrics. Figure 2.2 illustrates how the survey participants rated the importance in percent (y-axis) of various factors and artifacts (x-axis).

Further information needs can also be distilled from the various use cases that were mentioned by the participants. For instance, the collected information could help by identifying project artifacts that require additional testing or could be subject of refactorings. Also tracking the code stability over time would allow for more precise release planning [15].

From the developer's point of view, however, exists a serious need for identifying code changes that would require code reviews, due to their complexity or similar metrics. These reviews could also be used to evaluate the performance of the metrics retrospectively and adjust their thresholds, if necessary.

To date, there has been no such detailed survey of information needs at the portfolio level, but there exist some studies that deal with the problems and challenges that are related to software portfolio management. Based on these problem descriptions, the information needs that would be necessary to avoid or master these problems can be derived.

Card and Jones [60] conducted a survey among different companies and were able to identify various deficits: One of these major problems, among others, seemed to be a certain reluctance to end uneconomic projects within a portfolio and reuse the released resources elsewhere. This happens because there is not enough meaningful information on which an appropriate decision is to be based, especially since corporate policy interests often play a role here, too. A broader information basis, in order to be able to better assess the quality and future viability of a project, could possibly support the decision making process here. But even if an appropriate information basis is created, there

are further problems: Monitoring software portfolios over a longer period of time can generate a huge amount of data and there is a risk that project managers will not be able to identify and filter out the relevant data due to this information overload. And relying on incorrect or irrelevant data can be very problematic, as it could lead to wrong decision [60].

Various information needs can therefore be derived from these problems: First, there is a need to reduce the total amount of data to make them easier manageable, and at the same time the gathered information must also be of high quality, so that important decisions can be made based on them. Another study on software portfolio management comes to similar results, but emphasizes the additional need for so called *fire fighting*: This means that portfolio managers must be able to quickly recognize unexpected problems or events, so that they can react immediately [4]. This adds an additional demand for some kind of an automatic early-warning system to the pure information related needs mentioned by the other study.

Concluding this literature research, it can be said that numerous information needs of software quality managers exist and some of them can be very complex, affecting both, individual software, but also entire portfolio artifacts. At the same time, many different decisions are driven by the gathered information.

Needs that specifically focus on the visualization of the gathered information played only a minor role in the examined literature and were therefore further surveyed by additional expert interviews, as described later in Section 6.1.

2.4 Visualization Concepts

This section closes the Chapter Foundation by introducing some commonly used concepts for data visualization, together with their advantages and disadvantages.

According to [64], visual data exploration can be distinguished into three different phases, namely

- **Overview** At the beginning, the user wants to get a first overview of the data and also tries to identify possible points of interests.
- **Zooming and Filtering** During this phase, the scope of the data must be narrowed down and by using different filter options, noise and irrelevant data should be filtered out.
- **Details-on-Demand** After the relevant data is isolated, the user should now be able to drill down to a more detailed view where specific patterns and facts can be investigated in deep.

A data visualization system should not only support the user in all these steps by providing sufficient views, it must also be able to provide visualizations and functionality that connect the different steps with each other [64].

Before investigating some possible visualization techniques, a closer look at the data to be analyzed should be taken, as this will affect the visualization significantly. Depending on the number of variables of a single data record, data sets can be divided into one-, two- or multi-dimensional data [64]. In case of the visualization prototype, different scenarios can be possible, depending on the concrete use case: At the beginning, users might only want to investigate the historical trend of a single metric value over time. In that case, the snapshots correspond to specific points in time and the recorded metric values represents the measurements at these times. The resulting data set can then be interpreted as a one-dimensional time series. While this might be a valid entry scenario, most of the time users will probably prefer to combine several metrics to gather more insights. If more than one data value is available per time interval, the data could either be interpreted as multi-dimensional data or as a set of separated one-dimensional time series [65].

Depending on how the data will be interpreted, different types of views should be preferred for the visualization. Likewise, it can also happen that the user does not want to compare historical trends, but instead focus on the comparison of several metric values at a specific point in time. This would correspond to an extract of the aforementioned multiple times series and has to be handled separately.

As can be seen, different ways of interpreting the collected metric values are possible, and thus, there are also several ways to render the relevant data. The objective of the following section is therefore to present some of these visualization approaches, together with their advantages and disadvantages.

Scatter Plots and Line Charts are very commonly used for rendering time series, as they allow for easy recognizing a possible trend in the data series. Since, in case of time series, they are a two-dimensional projection of values over time, no more than one data point per time can be visualized per time series - as long as no other properties like color or shape are used to encode information. If more values have to be rendered, several time series have to be superimposed in the same diagram, but this may quickly lead to a cognitive overload and the user might loose track of the different series [65].

Another problem with line charts combined in one diagram is the scaling factor: Often, different time series might have different scaling factors (for instance, *lines of code* can grow up to several hundreds of thousands or more, while the *ratio to comment* value won't be more than 100 most of the time) and if both are rendered in the same diagram, the details of the smaller scaled line will get lost. To avoid this, instead of using absolute values, the relative changes compared to a predefined base line could be rendered instead, which would make trend comparison possible again [65].

Consider Figure 2.3: Both diagrams print the historical values of two metrics, *Lack of Cohesion of Methods* and *Number of Classes*. But since the latter one has a much larger value range, the LCOM trend (which values are between 0 and 1) would not be recognizable anymore. This effect can be seen in the left diagram where the y-axis is aligned to the value range of the *Classes* metric. Now consider the right diagram where

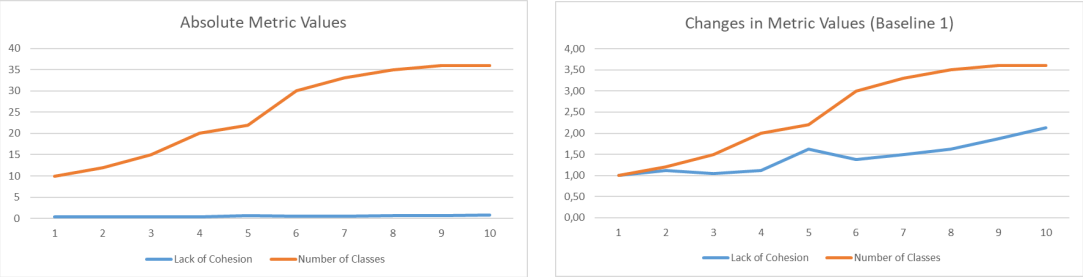


Figure 2.3: Both line charts illustrate the average *Lack of Cohesion of Methods* and the total *Number of Classes* for ten snapshots/commits. Because there is a large difference between the value ranges of these two metrics, the trend of the LCOM metric is not visible in the left diagram. The right diagram, in contrast, uses relative changes, which makes both trends recognizable again.

instead of the absolute values, the changes compared to the baseline (the first snapshot) are used instead. The LCOM trend now becomes visible and it can be seen, that while the number of classes reaches a specific limit, the cohesion becomes worse. This could be an indicator that too much logic was added to the system without separating the logic into different classes. But to further test this hypothesis, additional metrics (like, for instance, *Lines of Code per Method* etc...) should also be considered.

Histograms and kernel density plots (Figure 2.4) are best used to visualize the distribution of a specific metric value among the observed entities. Therefore, the possible range of values has to be split into intervals of identical or different ranges, and for every interval, the frequency of the values which lie within the interval boundaries have to be counted. Based on these two values, a rectangle can be drawn which area represents the frequency of the values measured within a given interval.

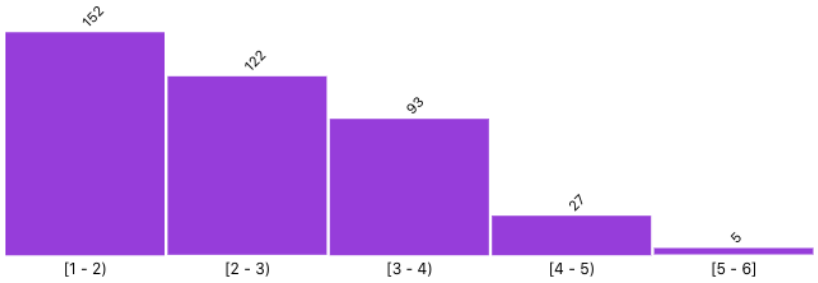


Figure 2.4: A histogram showing the distribution of numeric values. The total range of values is divided into several intervals and the measured values are assigned to their corresponding value ranges.

Analyzing the distribution of metrics can provide meaningful insights, but unfortunately, histograms have some disadvantages when it comes to visualization of multi-dimensional data or historical trends: Creating histograms which combine the value ranges of two or

metrics in one bin is theoretically be possible, but these would also be very hard to read because of the increased information density. Superimposing two or more histograms (one per metric) can also be problematic because then the different solid bars could overlap each other [65]. Rendering the historical distribution of a metric might also become complicated: Histograms are already two dimensional projections, so there is no easy way to add an additional dimension, like time, to the diagram, without creating too much information overload.

Iconic Displays, as depicted in Figure 2.5 use the properties of specific objects - like the dimension of a geometrical figure or the color of an icon - to encode different metrics [64]. In that way, up to five different metrics can be encoded by a single rectangle (x and y position, width, height and color) [66]. Health indicators can be considered as simplified icon displays, where only one variable - the metric value in comparison to a given threshold - is encoded. This could, for instance, be achieved by using a traffic light image that changes its color depending on the metric value. These types of displays could theoretically also be combined with other chart types to render historical trends, but this does also increase the cognitive burden of the user.

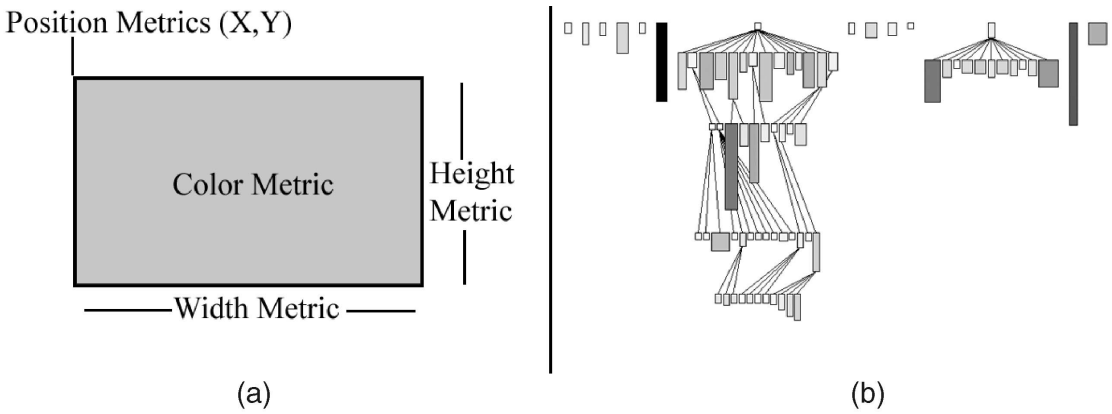


Figure 2.5: By mapping metrics to object properties, many different metrics can be encoded by a single geometrical figure (a). Combining several of these objects together allows the visualization of complex software architectures (b) [66].

Kiviat Diagrams (also known as polar- or radar charts) are well suited for displaying multivariate data, as they use one axis per variable and the variable value is mapped directly to the corresponding axis position. By connecting these different positions with each other, a geometric figure can be constructed which forms a unique representation of this specific variable constellation. This allows for easy recognition of specific patterns within the dataset (see also Figure 2.6).

By using these diagrams, it is also relatively easy to check whether each value of a data set is within a specific threshold: This would be the case if the geometry representing the observed values would be totally surrounded by the other geometry, representing the thresholds. Of course this will only work if all thresholds define either the minimum

or maximum value a variable should have. If the entries of the dataset use significantly different value ranges, scaling can become a problem because positions of smaller-scaled variables would be grouped around the center and could not be distinguished anymore. Normalization of the metric values can help in avoiding this problem [67], although the real values of the variables would get lost in that case. But since the diagrams are best used for comparing the geometrical patterns, this should not be such an issue. Feature vectors which should be represented by Kiviat diagrams need at least three components, otherwise, it would not be possible to draw a polygon by connecting the different axis positions.

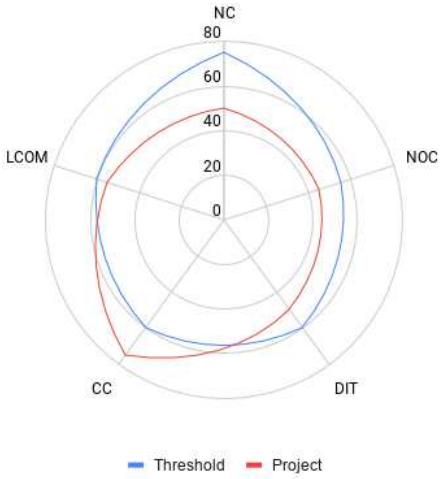


Figure 2.6: Metric values and their thresholds organized in a Kiviat diagram. It is recognizable that the measured cyclomatic complexity value violates the threshold, while at the same time, the system has a relatively low number of classes. This could be an indicator that too much business logic is concentrated in too less domain classes.

Tree Maps are a way to interpret software systems as hierarchical data: Projects are organized in modules or namespaces, and these are again organized in classes that, in turn, consist of methods and attributes [20]. One way of visualizing these hierarchical structures is by using *Tree Maps*, where a two dimensional pane is divided into separate chunks that represent the different levels of the hierarchical structure. The same procedure can also be used to visualize aggregated software metrics, and similar to *Iconic Displays*, a metric could be bound to more than one property to visualize multivariate data (see Figure 2.7). Navigating between different levels of details could also be provided intuitively by letting the user select a specific subarea (for instance, a rectangle that represents a class), and render a new tree map that shows the internal structure and metrics of the class members. While tree maps are a good tool for presenting hierarchical data, they also have some drawbacks: Depending on the underlying data, the tree structure can become very dense, and in that case, it is not always easy to clearly distinguish which area in the diagram belongs to which branch of the tree hierarchy. Voronoi Tree Maps [20], which

improve the tree layout by organizing the data points around a specific center point, are a possible solution for this problem, but their calculation and visualization is not trivial and not easily accomplished with off-the-shelve drawing components. Rendering historical trend data is also not easily achievable with tree maps, stacking different layers of them to represent different points in time would theoretically be possible, but might also increase the information overload significantly.

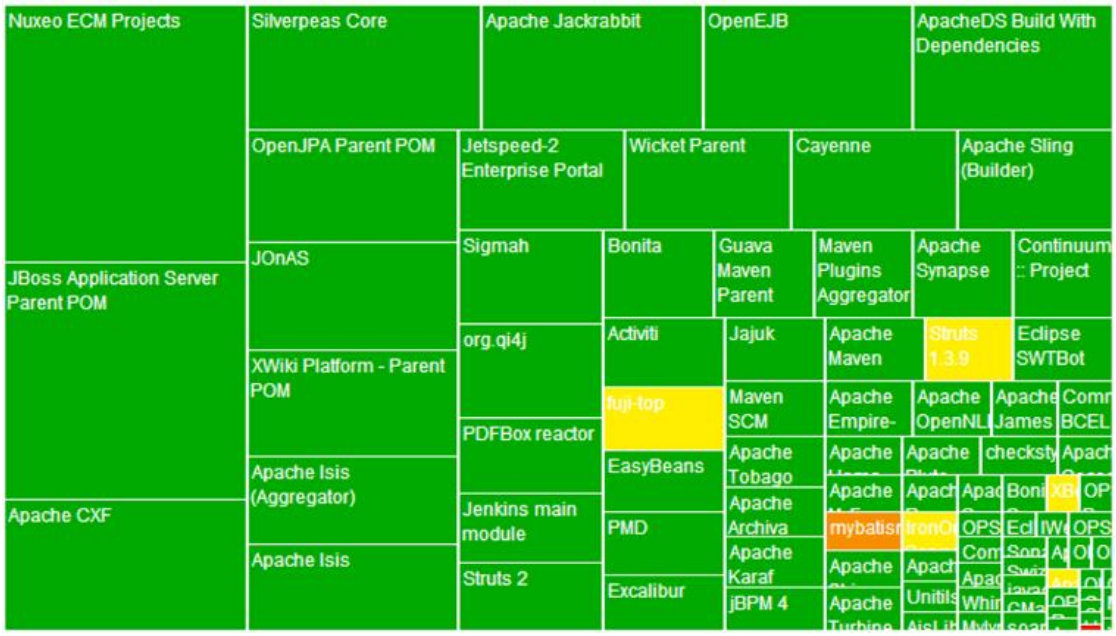


Figure 2.7: A tree map visualized by SonarQube. The size of the rectangles represents the lines of code per module while the color shows its overall rule compliance [68].

3D Diagrams Another way of visualizing software metrics is through the use of 3D diagrams, which provide an additional depth axis, and therefore can display higher dimensional datasets than pure 2D charts. Despite that, there are other interesting applications for use cases, for instance, *MetricView* [69] uses a combination of flat UML class diagrams together with an extrusive view that displays calculated metric values.

A similar concept was followed by Wettel and Lanza [70]. By combining the visualization of system architectures and metrics with topological concepts, they created *CodeCity*, a tool that uses city-like 3D models to represent software systems together with their quality metrics. Hereby, every building in the city represents a different class, and the properties of the building are specified by corresponding metric values, similar to the *Iconic Display* approach. Users can then "walk" through the system architecture and inspect possible points of interest.

One problem with 3D displays is the fact that, due to the limitations of our screens, they have to be rendered as 2D projections with a perspective distortion to simulate the third axis. This perspective drawing can make it difficult to compare geometries placed

on different depth levels [65], as objects in the background, even with the same height, might appear smaller than objects in the front. To solve this, the user must be able to freely adjust the view by rotating or translating the virtual camera that is used to render the scene. Although there exist various 3D libraries for most systems and browsers[2,3], implementing such an interactive user interface is often not a trivial task, and rendering complex 3D scenes might also require additional performance considerations [70].

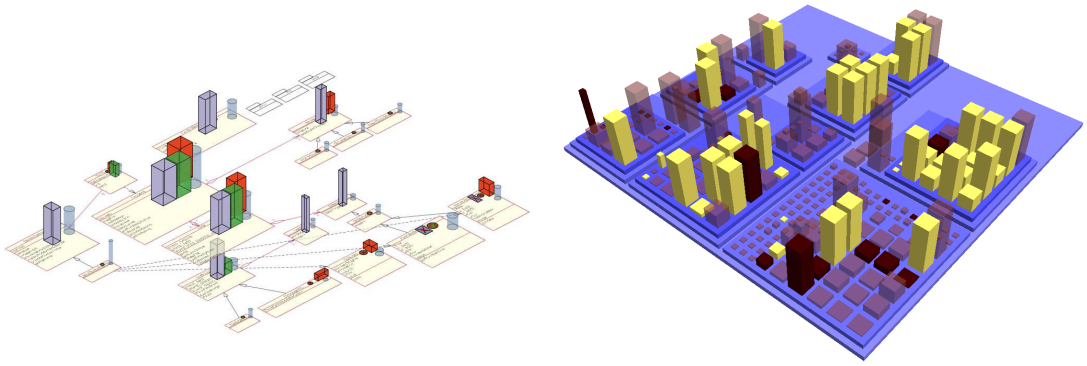


Figure 2.8: MetricView (left) uses 3D models to enrich UML diagrams with metric information [69]. CodeCity (right) generates a city-like topology out of classes and their metrics [70].

Besides the presented diagram types, there are numerous other visualization concepts like parallel coordinates [71], dense pixel displays like distance matrices [65] or stacked displays [64]. However, introducing all of them in detail would go beyond the scope of this work, and most of them would also require custom implementations, as many off the shelve chart libraries do not support these type of displays.

Visualization	Use Case	Pro	Contra
Line Chart	one dimensional time series	• trends are easily recognizable • multivariate dataset (by using more lines)	• risk of overplotting • all series need similar value range
Histogram	distribution of samples within a population	• shows frequency of metric values	• no multivariate data • no historical trends
Iconic Display	encodes values through object properties	• fast overview of value • easily recognizable	• only limited multivariate data support • requires thresholds or other criteria
Kiviat Diagram	visualization of multivariate data	• good for recognizing patterns • supports high dimensional data	• different scales must be normalized • at least three variables required
Tree Map	hierarchical data	• good visualization of hierarchical data • several metrics can be encoded	• visualization can become very dense • no easy visualization of historical data
3D Diagrams	versatilely applicable	• multivariate data support • view angle can be customized	• perspective distortion of data • rendering can be more expensive

Table 2.1: Various graphical concepts and diagram types

Table 2.1 gives a summarized overview of the advantages and disadvantages of the aforementioned visualization concepts. As suggested in [64], the visualization prototype

[2] `https://threejs.org/`, last accessed on 04.03.2020
[3] `https://www.babylonjs.com/`, last accessed on 04.03.2020

uses a combination of different views, depending on the concrete use case and level of detail the user wants to investigate. Despite tree maps and 3d diagrams, which would require a relatively high implementation effort, all other mentioned concepts were used in the final implementation of the prototype. While line charts (with one or more time series) were chosen for presenting historical metric trends, additional views like kiviat-diagrams or histograms are also implemented to support the user by gaining additional insights. Implementation details regarding the different views and the navigation between these will be discussed in section 6.4.

CHAPTER 3

State of the Art

Software portfolio management is a multifaceted area, and as such, also the subject of numerous research efforts. In addition to purely academic aspects, the topic is also of great economic interest for commercial companies, as for some, their business value is highly related to their own software portfolio [16], and managing these portfolios efficiently promises a competitive advantage over competitors. This demand also resulted in a large number of commercially or freely available tools for managing, analyzing and visualizing software portfolios.

The following chapter is therefore intended to provide an overview of current research results in various aspects of the topic and then presents some selected tools from the area of software portfolio management and visualization.

3.1 Current State of Research

Below enumeration of current research results is structured by the different subject areas from which the the main research question is compromised (see also chapter 4.1). For each topic area, the research survey most relevant for this work is briefly described, and it is also mentioned to what extent the underlying work was influenced by it.

Information Needs

In order to understand the problems and challenges associated with the visualization of software portfolios, it makes sense to first approach the situation from the perspective of the stakeholders, and hereby especially from the view of the software quality managers.

A comprehensive analysis of their information needs was carried out by Buse and Zimmermann based on a survey covering 110 software and lead developers employed at Microsoft [15]. Many of the insights gained from this study have been incorporated into

the requirement specifications (see section 6.2) for the prototype developed in this work. Based on existing sources, the authors first of all identified a considerable deficit between the information needs of these software managers and the information which is actually available. This deficit might even negatively impact the quality of software products within a portfolio.

Among other things, their investigation showed that software quality managers rely primarily on existing data and metrics, while software developers, in contrast, often make their decisions based on their personal experience. From this fact, the authors derive a certain basic need of the quality manager for analysis tools, provided that these tools are able to provide the required metrics.

Another important result of their survey is the realization that historical and current quality information about software projects are often more important for managers than forecasts on future development. This may be due to the fact that, on the one hand, these prediction models are relatively difficult to create and, on the other hand, the traceability of the gained results is not always fully transparent.

Regarding the level of detail of the project analysis, their study showed that although managers are primarily interested in higher-level project metrics, the possibility to analyze metrics at a more granular artifact level (such as files, classes, functions, etc...), is also highly desired.

Their investigation concludes with a catalog of suggested guidelines regarding the development of possible analysis tools. Many of these recommendations had direct impact on the implementation of the visualization prototype realized by this work.

Source Code Repository Mining

An investigation [16] has shown that software portfolios can consist of a large number of projects with several million lines of source code for each, which would make gathering and analyzing portfolio data manually practically impossible due to the enormous amount of time required. To make matters worse, software quality managers are usually busy with other tasks [17] and therefore can only invest limited resources on these processes. Therefore, to collect enough portfolio metrics feasible for further analysis and visualization, the development of a highly automated process is a necessity for data mining of software portfolios. But thanks to the large number of publicly available source code repositories (Github, for example, hosted more than 100 million repositories for the first time in 2018[1]) there has been a lot of research activity in this area in recent years, so that by now many different guidelines and strategies for implementing such a process are available.

The *MetricMiner* [37] is a very interesting project in this context, as it deals with a problem similar to the one underlying to this work: The authors provided the design and implementation of a mining tool chain for collecting and analyzing various metrics from different software repositories. For this, they first described a workflow (depicted in

[1] `https://github.blog/2018-11-08-100m-repos/`, last accessed on 22.01.2020

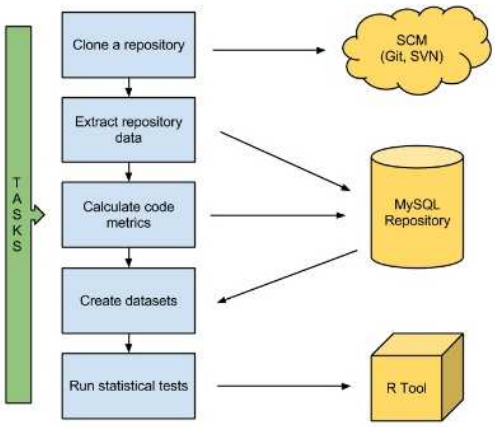


Figure 3.1: The workflow of MetricMiner [37]

Figure 3.1) consisting of the individual steps necessary to clone repositories, calculate metrics and run further analysis on the gathered data. The so collected data was then stored in an SQL database and could be accessed through a web interface for further processing. As different use cases require different metrics to calculate, an important point in the development of the *MetricMiner* was to make the system expandable in that way, that adding additional metric calculation providers at a later point in time would be possible. This extensible platform approach was also chosen for the repository mining process created during this work, as starting with only a few metric providers allowed for a more iterative development process, and the final decision which providers to use could be postponed to a later stage. While *MetricMiner* also supports the visualization of historical trends, it clearly focuses on the analysis of individual projects and not on overall portfolio overviews or in-portfolio comparisons.

A similar toolchain, called *GlueTheos*, is presented by Robles in [38]. This is another system for gathering and calculating software metrics from open source repositories. The architecture of *GlueTheos* is divided into different modules, each one responsible for a specific task during the mining process. This modular approach has the advantage of exchanging individual areas of the system and expanding it more easily, e.g. to query different types of source code repositories. In order to keep the implementation effort low, existing third-party tools, like external metric calculation providers, were integrated into the system where possible. Separating the architecture into different isolated modules was also partly adopted for the study at hand, as this allowed for a clearer and leaner structure of the mining tool chain.

In contrast to *MetricMiner*, *GlueTheos* supports comparing metrics of several projects with each other, even if this comparison works only on isolated snapshots and not on historical trend data.

Software Portfolio and Multi Repository Visualization

Even if the visualization of metrics within portfolios has not been the subject of scientific research that often, there do exist some papers that deal specifically with this topic, two of them will be presented here briefly. In addition, also studies were considered that focus on the visualization of metrics from different software repositories without assigning these projects to a specific portfolio. While in these cases the research question might be slightly different, the answers to this question can still provide insights that can be adopted for the present work.

A very extensive work on software portfolios was created by Kuipers and Visser as part of their methodology called *Software Portfolio Monitoring* [16]. The authors developed an iterative process for gathering, analysis and visualization of metrics obtained from software portfolios. Their process consists of three different iteration cycles, starting with monthly iterations, where metrics of different repositories were recorded (both automated and manually), analyzed and made available in the form of a visual report. The next higher, quarterly iteration summarizes the findings of these monthly iterations and should provide experts with enough information to reason about the overall condition and quality of the software portfolio. Finally, all gathered information should be presented during an annual iteration to the company's management. While the monthly and quarterly iterations might contain highly technical data and analysis, the yearly management report should take into account that the decision makers might not always have a profound technical background, and therefore the data should be prepared accordingly. For carrying out these three iteration cycles, the authors have developed a component-based framework, the *Software Analysis Toolkit* that handles the different tasks during the data mining process. Similar to [38], the system has a modular architecture and is separated into different parts, which are called components here. While the framework has - compared to other approaches introduced in this chapter - a much lower level of automation and depends largely on user interaction, it still has a much broader support for integrating visualization components than the other solutions presented so far. Beyond features like charts for showing metric trends and graphs for visualizing dependencies within the system, it also makes use of animations to express the historical development of several metrics. By applying animations as a visual feature (see Figure 3.2), the authors argue that they are able to present highly dimensional data more easily compared to simply using static charts, as here one of the axis would always have to be used to present the chronological course.

Collecting and visualizing high dimensional data from large portfolios also raises a number of additional challenges, one of these problems is information overload [2]: The mining process collects too much data from which the relevant information can no longer be extracted easily. Auer, Graser, and Biffl [39] try to deal with this problem by applying a method called *multidimensional scaling*, in which highly dimensional data is reduced to fewer dimensions in order to enable a better, explorative analysis. To achieve this, the different projects within a portfolio are interpreted as feature vectors where every feature describes a specific characteristic of the project. These vectors can then be further

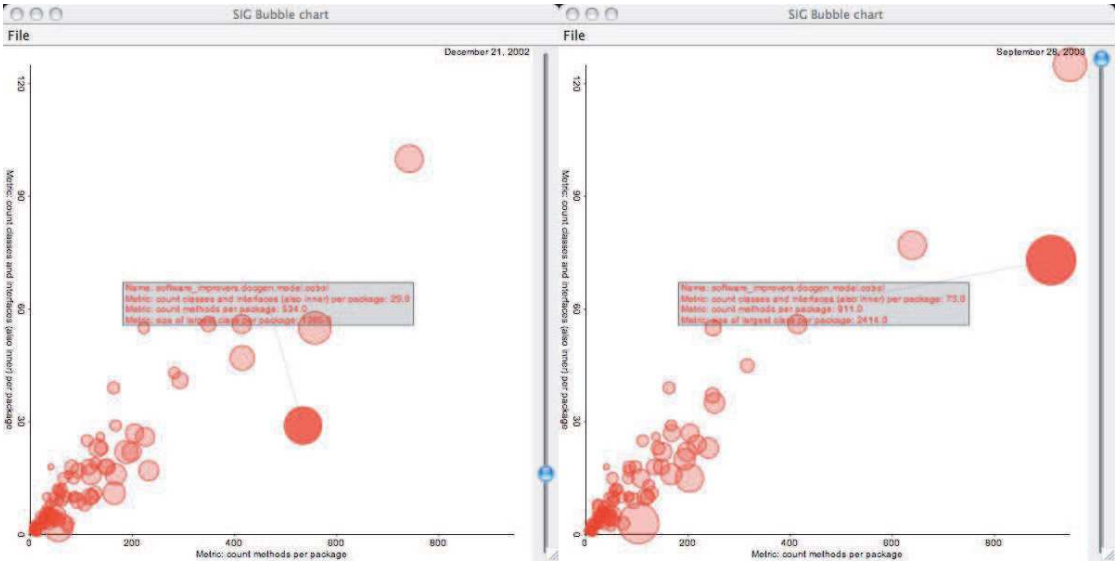


Figure 3.2: Software Portfolio Monitoring [16] uses animations to show the evolution of the package size of a Java system.

adjusted by applying different weight factors, depending on the relevance of the feature. Based on the adjustment of this weighting factor, the influence of certain factors on the overall portfolio can be isolated. What makes multidimensional scaling a very suitable approach for analyzing software portfolios is the fact that it is able to handle features with different scaling very well, a fact which can be encountered quite often since most software metrics use different value ranges. Nevertheless, this approach requires some knowledge of data science (the user must be able to alter the various parameters of the model and be able to correctly interpret the change), which cannot always be taken as granted regarding the target audience and therefore makes it not very suitable for this visualization prototype. This is aggravated by the fact that the time quality managers can invest in the analysis is often limited [17].

Two other research projects (Figure 3.3) do not explicitly deal with software portfolios, but describe holistic solutions for cross-project metric acquisition and visualization and should therefore be also briefly introduced here.

The *Empirical Project Monitor* collects data from various repository sources (source code repositories, bug tracking databases and mailing lists) and provides the user with a view of the interlinked data [40]. This makes finding connections and relations between different artifacts not only much easier, but also gives the user a better overview of the overall context of specific events (like for example mapping incoming bugs and their fixes to relevant knowledge exchange via e-mail). In addition, the metric trends of several projects can be compared with each other and also their distributions across individual projects can be compared. In total, the *Empirical Project Monitor* supports three different types of graphs and provides also a plain SQL interface to let users query the database

manually. The authors also identified the problem of information overloading when displaying high-dimensional data, but do not offer a solution for this in their survey.

Another work describes the development of the so called *MetricViewer*, a web-based user interface client that is connected to a service-oriented architecture responsible for the acquisition of different software metrics [41]. The underlying system offers several web APIs for mining repositories and calculating metrics, but the visualization is finally done by Google's Chart API [2] - a web service for creating diagrams and graphs. The web services can be accessed via an abstract API interface that must be provided by their concrete implementations. For example, the API interface for repository mining (SO-MSR) defines a number of basic functions that are necessary to download external source code repositories, checkout specific revisions and provide meta data for these revisions during the mining process. Parts of this interface specification were also adopted to define the functional scope of the repository mining interface in the present prototype (see chapter 5). As XML - which is quite hard to read for humans - was chosen as exchange format between the services, an additional graphical user interface was developed to simplify the human interaction with the system. The visualization of this graphical client uses a very fine-grained approach, by providing most metrics on file level. Project wide overviews or comparisons between projects are not supported. This file-based representation has partly influenced the visualization of individual artifacts in the present work, since this approach does offer a possibility to satisfy the information need of analyzing isolated elements of a project, as mentioned by [15].

[2] `https://developers.google.com/chart`, last accessed on 23.01 .2020

Figure 3.3: Empirical Project Monitor [40] showing the comparative trends of selected metrics (left) and the dashboard overview used in MetricViewer [41] (right)

3.2 Available Tools

In addition to pure research work, there exist also a wide range of commercial tools for analyzing and visualizing software portfolios. Depending on their concrete use case, these tools differ greatly in their functionality, handling and also pricing range - although free versions are often available for non-commercial use. The following list gives an overview of the most common applications from this segment. Tools which have a more isolated focus, like pure metric calculation providers, or of a more broader use case, like complete continuous integration systems such as Azure DevOps[3] or Jenkins[4], are not considered here.

SonarQube/SonarCloud

These two products from SonarSource[5] are static code analysis tools that are either available as cloud service (SonarCloud[6]) or as on-premise solution (SonarQube)[7] and are therefore best integrated directly into a complete continuous integration process. As both platforms provide a quite similar range of functionality, the following description will focus on the on-premise version SonarQube. Its community edition is freely available and also the sources are distributed under the GNU Lesser General Public License[8]. Besides this free edition exist also some commercial versions with varying features, mainly targeting at companies of different sizes. SonarQube's pricing model is based on used instances and lines of code to be analyzed, versions with more features and additional support, like the Enterprise edition, can easily account for several thousands Euros per year[9].

Due to its plugin-based architecture, SonarQube supports a wide variety of different programming languages and types of source code repositories. The source code analysis focuses on static code analysis rules for tracking technical debt and code security issues or code smells, but there are also a few metrics calculated during the analysis phase, like single lines of code, cyclomatic complexity or test coverage rate. Thresholds for these metrics can be defined by so called *quality gates*.

To express the health status of a project, SonarQube uses a grading system, similar to the ones which are used in schools. This makes the analysis results more comprehensible for users with a less sophisticated technical background. Using this kind of discrete evaluation scale instead of continuous values per metric has become quite popular in software analysis tools and can also be observed in some of the other applications presented here. A build that is supervised by SonarQube is classified in terms of fulfillment of the configured quality gate and allows the definition of actions in case of non-compliance. Different

[3] https://azure.microsoft.com/en-us/services/devops/, last accessed on 01/24/2020
[4] https://jenkins.io/, last accessed on 01/24/2020
[5] https://www.sonarsource.com/, last accessed on 24.01.2020
[6] https://sonarcloud.io/, last accessed on 24.01.2020
[7] https://www.sonarqube.org/, last accessed on 24.01.2020
[8] https://github.com/SonarSource/sonarqube, last accessed on 24.01.2020
[9] https://www.sonarsource.com/plans-and-pricing/, last accessed on 25.01.2020

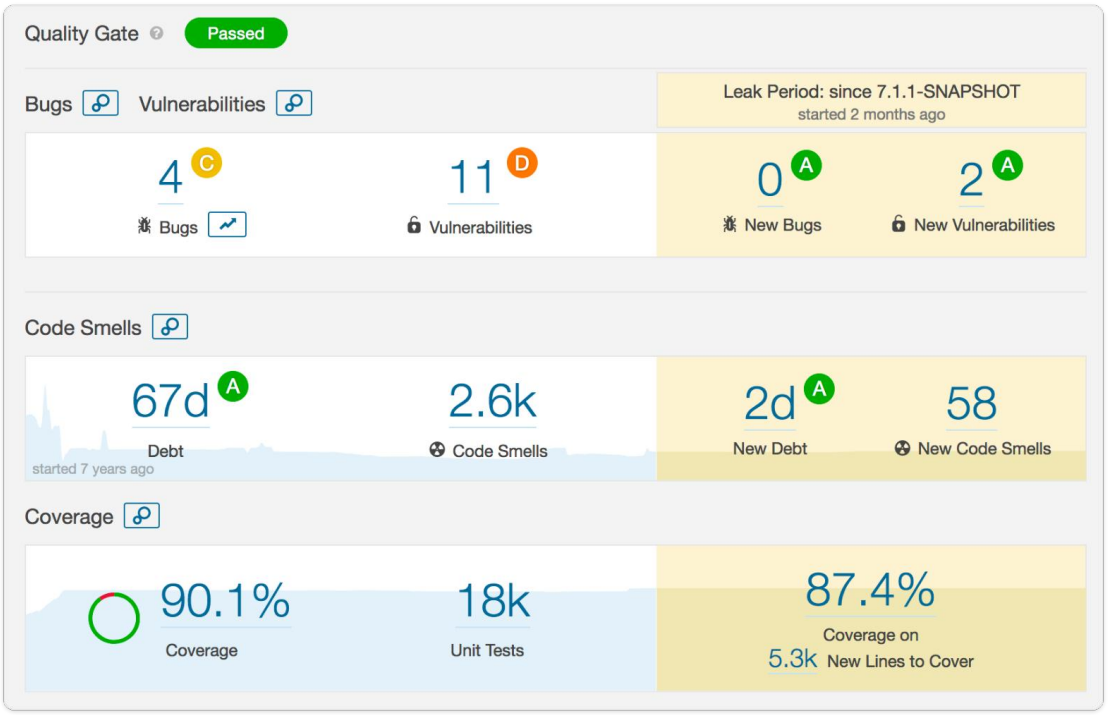


Figure 3.4: SonarQube Dashboard[10]

dashboards (see Figure 3.4) provide an overview of the current quality states of selected projects and allow users to jump directly to the line of source code that causes possible quality issues.

While there is the possibility to view historical data per single project view, there is no direct way to compare the metrics of different projects with each other in an integrated view. The enterprise version of SonarQube provides support for visualizing software portfolios - as can be seen in Figure 3.5 - however in this case, the visualization is limited to snapshots and cannot be extended to compare the evolution of portfolios.

Codacy

Codacy[12] is an automated code analysis and review tool. It is developed by a company with the same name and is available either as software as a service platform or as on-premise solution. Its function scope can be compared to the one offered by SonarQube and it also supports so called *Quality Settings* that define thresholds for determining the health status of a project, similar to the quality gates used in SonarQube.

[10] `https://medium.com/red6-es/go-for-sonarqube-ffff5b74f33a`, last accessed on 24.01.2020

[11] `https://www.sonarsource.com/plans-and-pricing/enterprise/`, last accessed 24.01.2020

[12] `www.codacy.com`, last accessed on 24.01.2020

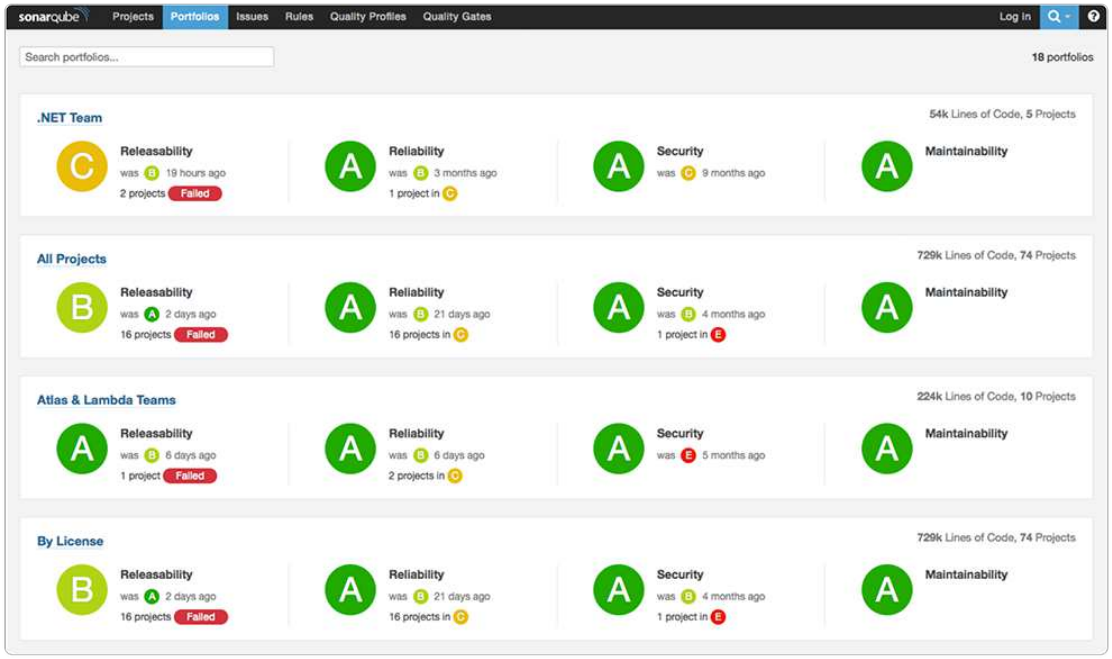


Figure 3.5: SonarQube Portfolio view[11]

Besides the different metrics, a grading system ranging from A to F is used to express the health status of both, single artifacts (like files) and overall project state. Connecting to a source code provider like Github and analyzing one's repositories works mostly via automated web API access and does not require much human interaction.

Analysis capabilities of Codacy seem to be a bit limited compared to SonarQube: Trend analysis is only available for the last month (or the recent eight commits) and apart from the dashboard overview, there seems to be no way to compare the metrics of different projects with each other directly.

Codacy's pricing model is currently based on three different plans, where analysis of open source projects is free of charge and commercial projects are billed based on users per month[13], making it particularly interesting for smaller businesses or start-ups.

Code Climate

Code Climate Inc. offers a product suite of software quality analysis tools[15]. *Velocity*[16], one application in their portfolio, is mainly targeting on the needs of project managers and decision makers, but focuses more on project and team related metrics like throughput,

[13] `https://www.codacy.com/pricing`, last accessed 25.01.2020

[14] `https://medium.com/@slryit/codacy-an-easy-to-use-code-quality-review-solution-5daf876d`
last accessed on 24.01.2020

[15] `https://codeclimate.com/`, last accessed on 25.01.2020

[16] `https://codeclimate.com/velocity/`, last accessed on 25.01.2020

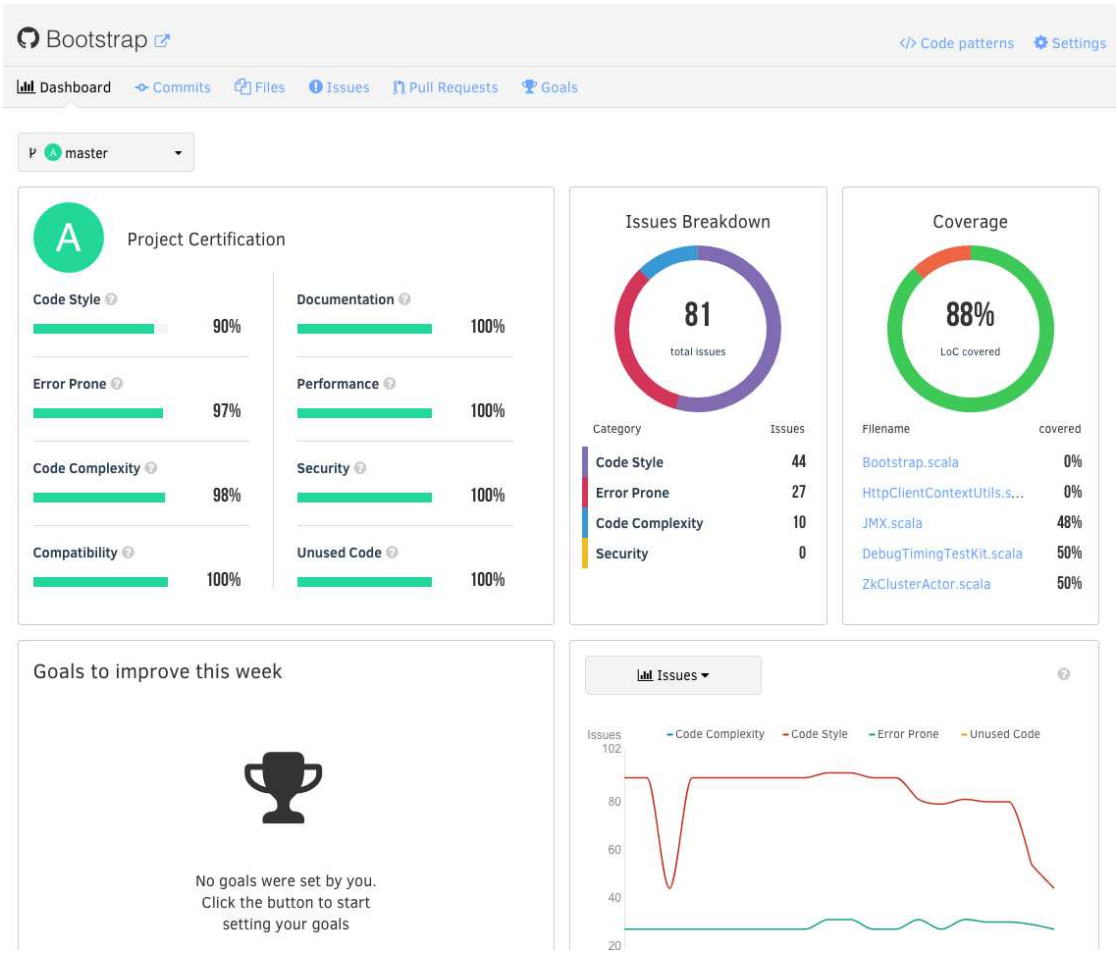


Figure 3.6: Codacy[14]

cycle time, pushes per day and others, to evaluate possible bottlenecks during the development phase.

Quality[17] (see Figure 3.7) on the other hand is an automated review and quality tool and is therefore more comparable to tools like SonarQube or Codacy.

It allows the analysis of source code repositories and calculates common metrics like code duplicates, code smells or test coverage. Also, an overall maintainability index and a file-based index are calculated and rated on a scale from A to F, similar to the ones available in Codacy. Trend overviews of technical debt or lines of codes are available, but direct comparison of different projects at the same time is not supported.

The pricing structure of *Quality* is also comparable to the one defined by Codacy and

[17] `https://codeclimate.com/quality/`, last accessed on 25.01.2020
[18] `http://mattaningram.com/work/codeclimate/`, last accessed on 25.01.2020

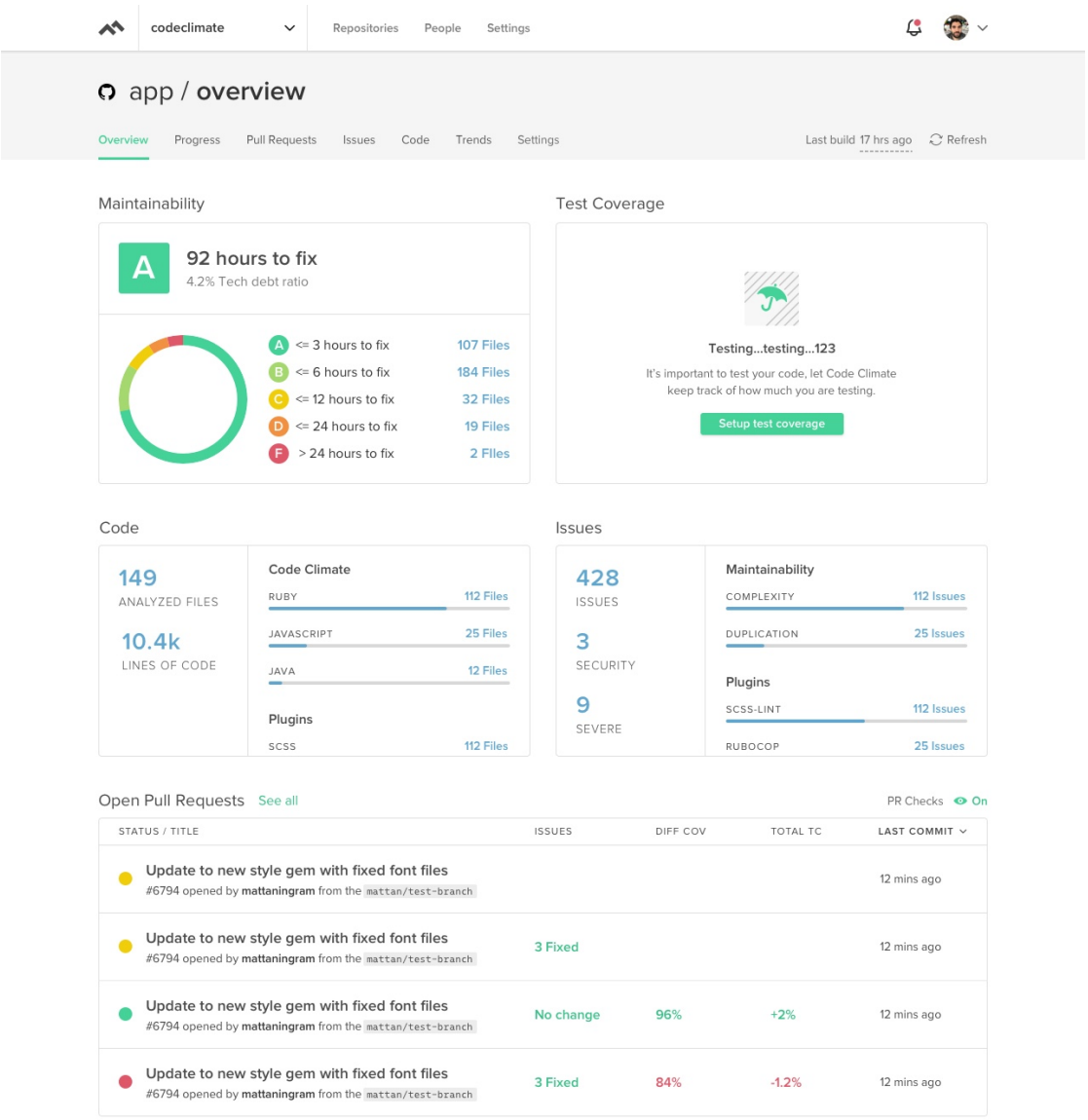


Figure 3.7: Code Climate's Quality[18]

might mainly be targeted on start-ups and smaller project teams, while *Velocity* is probably more oriented on enterprise customers, as for larger teams, costs can accumulate to several thousands of dollars per month.

GrimoireLab

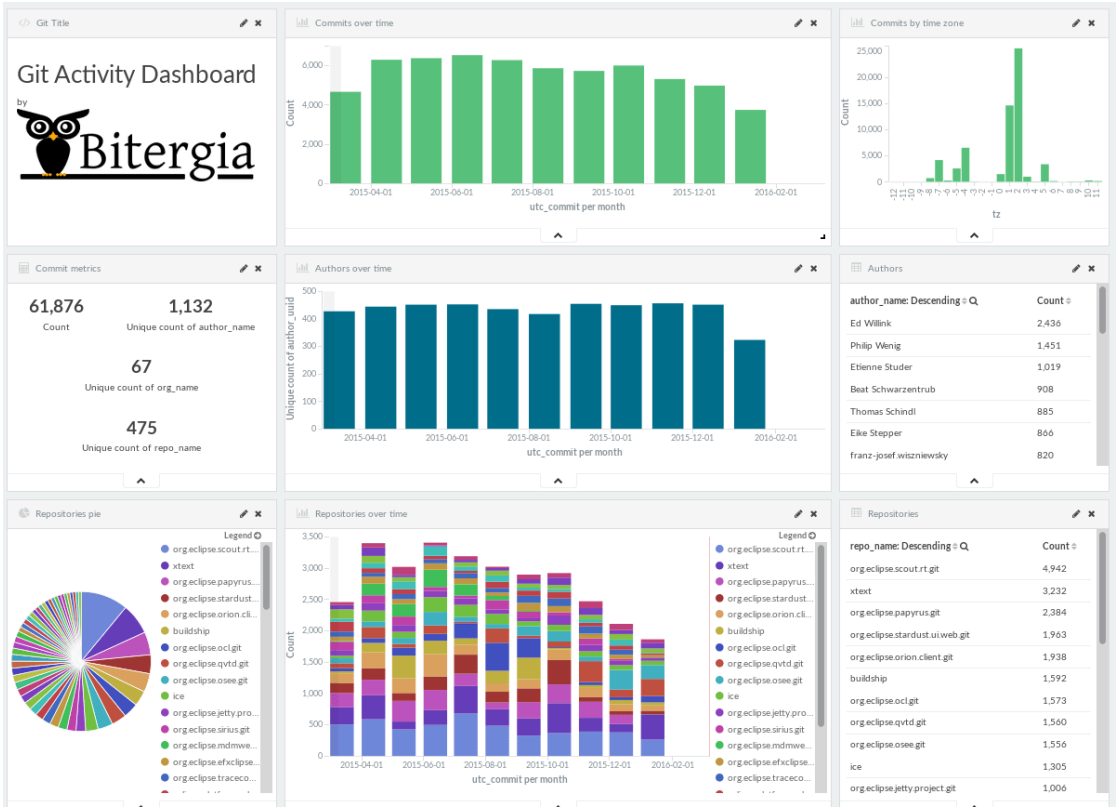


Figure 3.8: GrimoireLab for eclipse community[19]

GrimoireLab[20] plays a bit of a special role in this list: First, it is part of the Linux Foundation CHAOSS Software community[21], and as such, it is not a commercial product but freely available under the GNU General Public License. And second, its focus lies more in community analysis than in software quality analysis. Apart from that, its modular, component-based architecture makes it an interesting technological showpiece. It is comprised from several components, each one responsible for a different task:

- *Perceval*[22] is responsible for collecting data from different repositories, like version control systems, bug trackers or mailing lists.
- *Arthur*[23] on the other side orchestrates and manages the Parceval component to allow for parallel downloads and data retrieval.

[19] https://blog.bitergia.com/2016/03/09/dashboards-for-the-eclipse-community/
[20] https://chaoss.github.io/grimoirelab/, last accessed on 25.01.2020
[21] https://chaoss.community/, last accessed on 25.01.2020
[22] https://github.com/chaoss/grimoirelab-perceval, last accessed on 25.01.2020
[23] https://github.com/chaoss/grimoirelab-kingarthur, last accessed on 25.01.2020

- *Sorting Hat*[24] is a component used to track identities over different data repositories.
- *Kibiter*[25] and *Sigil*[26] finally are responsible for visualizing the gathered data in a widget based user interface.

While this modular approach makes the system highly configurable, this configuration might also require expert knowledge at some points and certainly also a significant amount of time and resources, which might be a considerable high entry barrier for some quality managers.

Others

There are some applications that differ in their functionality or scope from those presented so far, but in one way or another still provide quality analysis and should therefore be briefly also mentioned here. Particularly noteworthy in this context are some Integrated Development Environments (IDEs) which, in addition to their function as programming tools, also offer various options for static code analysis. For example, Microsoft's Visual Studio[27] offers calculation of various object-oriented and complexity metrics on all projects of a solution, and similar plugins also exist for other IDEs, like IntelliJ from JetBrains[28]. However, these integrated solutions primarily pursue a different use case, as they try to support software engineers during the development process and give immediate feedback about the current health of their software projects.

Software portfolio analysis tools, on the other hand, are more used to track and analyze quality and health metrics of entire software portfolios over a longer period of time and by this support the process of making strategic decisions concerning not only isolated projects, but also whole portfolios.

3.3 Summary

An overview of the current state of research has shown that the management and visualization of software portfolios is a broad research area with many different focuses. While there are several papers dealing with repository data mining [37, 38], surveys investigating the interactive visualization of this gathered data are more sparse. It is interesting to note in this context that most of these studies are usually limited to one or two dimensions - they either compare only snapshots of multiple repositories [39] or analyze the trend metrics of single isolated repositories [41] - but comparison of metrics

[24] https://github.com/chaoss/grimoirelab-sortinghat, last accessed on 25.01.2020
[25] https://github.com/chaoss/grimoirelab-kibiter, last accessed on 25.01.2020
[26] https://github.com/chaoss/grimoirelab-sigils, last accessed on 25.01.2020
[27] https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019, last accessed on 26.01.2020
[28] https://blog.jetbrains.com/idea/2014/09/touring-plugins-issue-1/, last accessed on 26.01.2020

from several repositories over a longer period of time rarely takes place, and if it does, it usually focuses only on a few metrics like in [40].

A similar picture can also be seen with the examined software portfolio analysis platforms. SonarQube and GrimoireLab are among the few that enable analysis and comparison across entire portfolios, but especially with SonarQube trend analysis over historical portfolio data is not supported.

Most of these tools also provide a similar selection of metrics, usually single lines of code (SLOC), code duplicates, test coverage and some others. Besides that, these metric views are often predefined and the user cannot easily customize the selection of metrics she or he wants to compare with each other. While some of these applications could be extended by custom plugins, the development of these extensions might take a considerable amount of effort and will not always be feasible, as quality managers can often be quite busy [17]. GrimoireLab is an exception here, since it focuses on the analysis of open source communities and accordingly, provides other, community related, metrics.

3.4 Distinction from Current Research

The present work is built on some aspects of the presented studies and also uses them as inspiration, just as some features of the prototype are based on the existing tools. Nevertheless, this work separates itself in some parts from the current situation and adds new contributions to the scientific discussion, some of these differences should be pointed out in the following section.

Starting with the analysis of quality manager's information needs, current research mainly focuses on single project analysis [15]. While there do exist studies at the portfolio level [2, 16], they mainly focus on identifying organizational problems and less on concrete information needs. But based on current knowledge, a cross-project and portfolio-wide analysis concerning the visualization demands in quality aspects, as carried out in this work, is not yet available.

The situation is similar with previous work on repository mining and project or portfolio visualizations: Regarding the possible available dimensions - (1) historical trend data, (2) variable number of metrics and (3) number of investigated repositories within a portfolio - most studies in this area focus on only two of the three aforementioned. There are studies investigating many metrics of various projects, but only for a specific commit [38], and other studies present visualizations for historical trends, but here portfolio-wide comparisons are not supported [37]. An approach, as the one described here, that focuses on historical trends within a whole portfolio, which also regards visualization aspects, has not not yet existed so far.

Looking at introduced quality analysis tools, the situation does not look much different: From all examined applications, SonarQube probably offers the largest feature set regarding project quality analysis, and, in contrast to most of it competitors, it also

provides portfolio support, although some experts mentioned they miss cross-project and portfolio wide comparison features here (see section 6.1).

Concerning historical trend analysis and the used continuous integration (CI) process, tools often do not analyze all possible snapshots but focus more on build intervals. Here, several snapshots could even be combined into a single build, again depending on the configured CI solution. In some cases, this can go so far that quality metrics are only collected for the nightly builds, once a day. In contrast to this practice, a snapshot- or commit based sampling rate would be preferred, as was revealed during expert interviews (again, see section **??** here for more details). Therefore, the prototype uses a snapshot-based frequency rate at which the metrics are surveyed.

CHAPTER 4

Methodologies

The methodological focus of this work lies in the creation and implementation of a prototype for an expert visualization system, together with a scenario-based expert evaluation. The prototype consists of two separate parts, a mining toolchain for collecting all relevant quality metrics and an interactive visualization system for presenting the data to the end user. Regarding the chronological order, the mining toolchain was developed first, and only afterward, the visualization system was implemented. This is also reflected by the work's structure, where the implementation process of both components is described in two separate, consecutive chapters, following mostly the same methodological approach described here. All additional methodological steps and activities which accompanied this prototype implementation and evaluation are described in this chapter.

4.1 Research Question

Following the approach recommended by Siddaway [19], in a first step, the scope of the available research articles was narrowed down by formulating a concrete research question. The research area identified in that way was further broken down into different subtopics, each of it dealing with a specific sub-aspect of the overall question. The topics identified here were (1) software portfolios and the various aspects that distinguish them from single repositories, (2) the mining of software repositories to gather quality metric data, (3) the characteristics and analysis of complete historical quality metric trends, and finally (4) the interactive visualization and presentation of these quality trend metrics to end-users. Figure 4.1 shows these different subject areas that could be identified during this process.

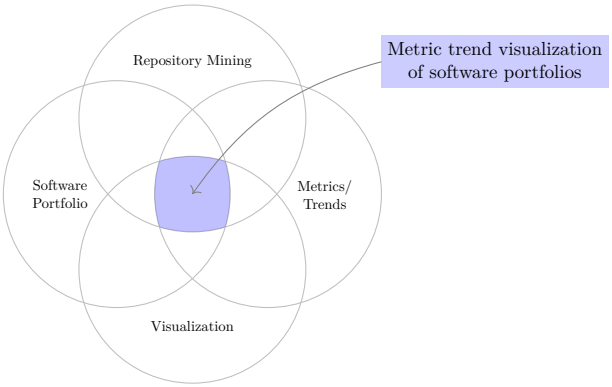


Figure 4.1: The different research areas that comprise the research topic. The specific research question focuses on the intersection of the individual areas.

These different sub-areas were then used as starting points for further examinations [19].

4.2 Systematic Literature Review

On the basis of the previously identified topics, a keyword-based systematic literature search was carried out, which was mainly oriented towards the following two goals:

1. Identifying any studies that already deal with the present research question in this or any similar form and
2. finding a selection of possible research papers that could serve as a basis for further investigation.

The search engines that were primarily used for the keyword search were *Google Scholar*[1], *Microsoft Academic*[2], *IEEE*[3] and the *ACM Digital Library*[4] as well as the integrated search engine of *Mendeley*[5]. To ensure that the search queries deliver the most representative result possible, different synonyms were used for individual terms [19]. Also, various search terms were used, including both compact terms such as "OOP metric" or "software portfolio", but also more complex expressions like "metrics calculation over historical data". In some cases, similar terms that were suggested by the search engines were also added to the search list. The following terms and their variations were used for the literature search:

[1] https://scholar.google.at/
[2] https://academic.microsoft.com/home
[3] https://www.ieee.org/, last accessed on 29.09.2020
[4] https://dl.acm.org/
[5] https://www.mendeley.com

Overall Concept "software portfolio visualization", "software portfolio analysis"

Software Portfolios "software portfolio", "software portfolio management", "software project managers information needs", "quality manager information needs"

Repository Mining "software repository mining", "software repository mining tools", "multiple project data mining"

Metric Trends "metrics calculation over historical data", "OOP metrics", "software metrics tools"

Visualization "software portfolio visualization", "portfolio visualization", "information visualization", "software metric visualization"

A considerable number of papers were found that dealt with the individual topics, and there also exist research work in which some of the above-mentioned subject areas overlap, but none of the identified surveys examined the specific research question as a whole. So does [11] for example compare the metrics of different source code repositories, however no historical data is included in the analysis here. Similarly, there are also papers that focus on the visualization of metrics [20], but here the studies are mainly limited to individual repositories and not to the comparative representation of several repositories as part of a software portfolio. Chapter 3 provides a summary of the found and relevant research and emphasizes in particular the similarities and differences to the present work.

The PRISMA workflow (depicted in Figure 4.2) according to [21] was used as a basis for further review of the existing literature. The examinations were considered in an iterative process and either selected or excluded according to defined criteria. The following criteria were mainly used to include/exclude papers:

Relevance Based on the abstract, a rough assessment of the topic of the paper was made. If it became apparent from this that there was no clear relation between its subject area and the research question of this work, it was excluded. If this question could not be clarified beyond doubt on the basis of the abstract, the paper was considered for further investigations.

Date of Publication Newer publications were preferred and publications that were older than 25 years were generally not considered, unless they provide general necessary knowledge of or were of particular relevance to the problem (this includes work like [22], which for example contains some basic definitions in the area of object oriented metrics).

Seriousness/Respectability This decision criterion is certainly subject to a certain subjectivity and was therefore only used as a support, but usually not as a decisive factor. The respectability of a paper was determined on the basis of the topic, the scientific writing style, the correct use and indication of references, as well as on the basis of the conference at which the paper was published.

Papers selected in this way have now been subjected to a closer examination following a three pass approach described by [23]. During this process, the paper was analyzed by applying different levels of detail, where the first pass concentrates only on information like the abstract, section titles and diagrams/figures and the second goes into more details regarding the key aspects of the paper. Since the third pass, as suggested by [23], is mainly relevant for paper reviewers, it was often omitted during the validation process.

If it has been shown that the paper was of no further relevance for the investigation, this was recorded with a corresponding note (in order to facilitate later evaluation due to possible referencing). Based on this first core of publications, further referenced articles have now been identified and analyzed according to the PRISMA workflow, see also Figure 4.2. In addition, the analysis of a paper could also lead to possible new keywords that seemed to be suitable for a further literature search. In this case, another literature search was carried out with the new search terms and the results were analyzed accordingly.

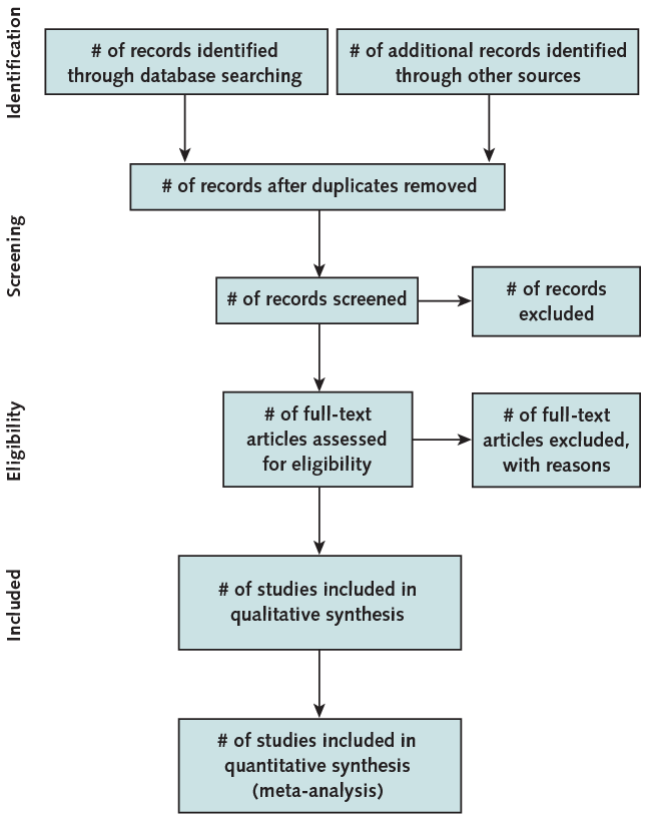


Figure 4.2: The PRISMA workflow describes an iterative process for systematic literature researches [21]

4.3 Technological Review

During this work, several technologies had to be evaluated for a variety of purposes, including in particular:

Existing Software Portfolio Analysis Tools This includes in particular applications that compete regarding functionalities and use cases with the prototype to be developed. The most relevant applications (on-premise, but also software as a service (SaaS) or cloud solutions) will be described in more detail in Chapter 3.

Metric Calculation Tools Since the calculation of metrics can be time consuming and complex, off-the-shelve implementations are preferred. The selection criteria and the possible tools are presented in Section 5.3.

Programming Languages, Libraries, Frameworks These are all technologies necessary to implement the prototype, starting from backend technologies like a suitable database up to user interface components for the frontend. The used technologies are described in detail in the corresponding chapters.

Wherever possible, existing literature was used to select the technologies (e.g. when comparing different tools for determining software metrics [14]), in most cases, however, a selection was made on the basis of internet searches, taking into account various criteria, among others:

- Is the technology freely available or is there a freely available license for research purposes?
- Is the use of the technology restricted by possible license terms?
- In case of third-party components, how well can they be integrated into the existing system?
- Is there sufficient documentation, do there exist sample applications?
- Is the technology actively developed or has its support already been discontinued?

4.4 Requirement Analysis and Specification

Requirements of the two aspects of the prototype (repository mining tool chain and expert visualization) were elicited separately.

For the mining tool chain, existing research, that focuses around the design of similar systems, was examined and based on these findings, the key features and characteristics of the prototype were specified. An explicit distinction between functional and non-functional requirements, as suggested by [24] was not made in this case since this separation did not appear to be necessary for the prototype development.

Since identifying and addressing the information needs of quality managers makes up an essential part of this work, and most features of the visual prototype direcly relay on these needs, a more detailed approach had to be taken here: By first defining the concrete role of software quality managers as stakeholders [24], a literature-based search was used to identify the information needs of these specific group of users. Since not all questions could be answered conclusively by pure literature research, additional information was collected through expert interviews [25].

Although the interviews were done in person, an online tool[6] was used to initially design the questionnaire, as this allowed for a more iterative and collaborative workflow. The selected group of experts was chosen to be relatively small (one participant during the pilot phase and three for the actual interviews, where the pilot participant was also a member of the later expert group) and therefore a qualitative survey was preferred against a quantitative one. The design of the questionnaire was based on guides suggested by [26] using both, closed questions based on a five-point Likert-scale [27], but also open questions. While these open questions might hold the risk of misinterpretation by the participant [26], possible misunderstandings could be discussed during the interview and due to the limited number of survey attendees, the manual evaluation effort compared to closed questions remained manageable.

In a first pilot phase, the questionnaire was reviewed by an expert and was further refined based on her suggestions. This was done to identify any issues with the structure and design of the questionnaire itself [26], evaluating possible answers to the questions was not part of this pilot study. The final survey was then used to carry out one-to-one personal expert interviews.

Evaluation of the questionnaire was done by applying *Thematic Analysis* [28], a systematic approach that aims at identifying patterns and themes within a data set and tries to draw conclusions based on these findings. This analysis process can be separated into six subsequent steps [28]:

1. **Familiarize with the data** Before starting the analysis process, one should actively read through the whole data set and reflect about the reading. Making additional notes about interesting or unexpected aspects of the data might help for further analysis.

2. **Generate codes** A code can be considered a label that defines a small underlying concept of a specific text passage. These could either be specific semantic information that can directly be extracted from the data set, but interpretations made by the researcher are also possible here. Ideally, codes should be formulated in that way that their meaning could be recognized and understood independently from the underlying data set.

3. **Search for themes** In this step, similar codes should be clustered together to larger, coarse grained patterns, called *themes*. Themes should span over all data

[6] `https://www.google.com/intl/de_at/forms/about/`, last accessed on 21.02.2020

records and should be related to a central idea that is related to the research question.

4. **Review potential themes** After the different themes were identified, their relation to each other and to the overall concept should be further refined. Furthermore, it should also be verified that the themes are based each on enough sophisticated entries from the data set.

5. **Define and name themes** During this process, a good name and description should be found for a theme. The description should summarize the overall focus and scope of the theme, similar to a research paper's abstract.

6. **Produce report** All insights that were gained during the previous five steps should be included in the final report. The themes should be presented in a logical and meaningful way and depending themes should build on each other.

The aforementioned six steps were all applied on the transcriptions of the expert interviews. Due to the small sample size, the results cannot be considered representative for the entire target audience, but were still helpful in better defining the application's scope and also supplemented the findings that were gathered during the literature research.

4.5 Development Process

4.5.1 Project Organization

An iterative and agile software development workflow was chosen to implement the prototype. For this, the overall work was first divided into different sub areas (one per chapter), and these areas were then further broken down into individual sub tasks. Each sub task was gradually refined until its description contained enough information to allow for a concrete implementation. All tasks were organized in a four-column online task board (see Figure 4.3) and every work item had to go to the following states during the development process:

1. **Todo** This is the initial column where all new work will be captured first. While tasks in this column should already have a rough description, their concrete requirements could also be specified at a later stage. It could also happen that during development, a task in this column became obsolete due to changing requirements or because of new insights, in that case, the task was simply removed from the board.

2. **Next** This column serves as a pre-sorting mechanism to allow for better estimating upcoming workloads. Whenever a task is placed in this category, its requirements should be formulated so precisely that work on it could be started during the next iteration. If during analysis a task proved to be too difficult for implementation

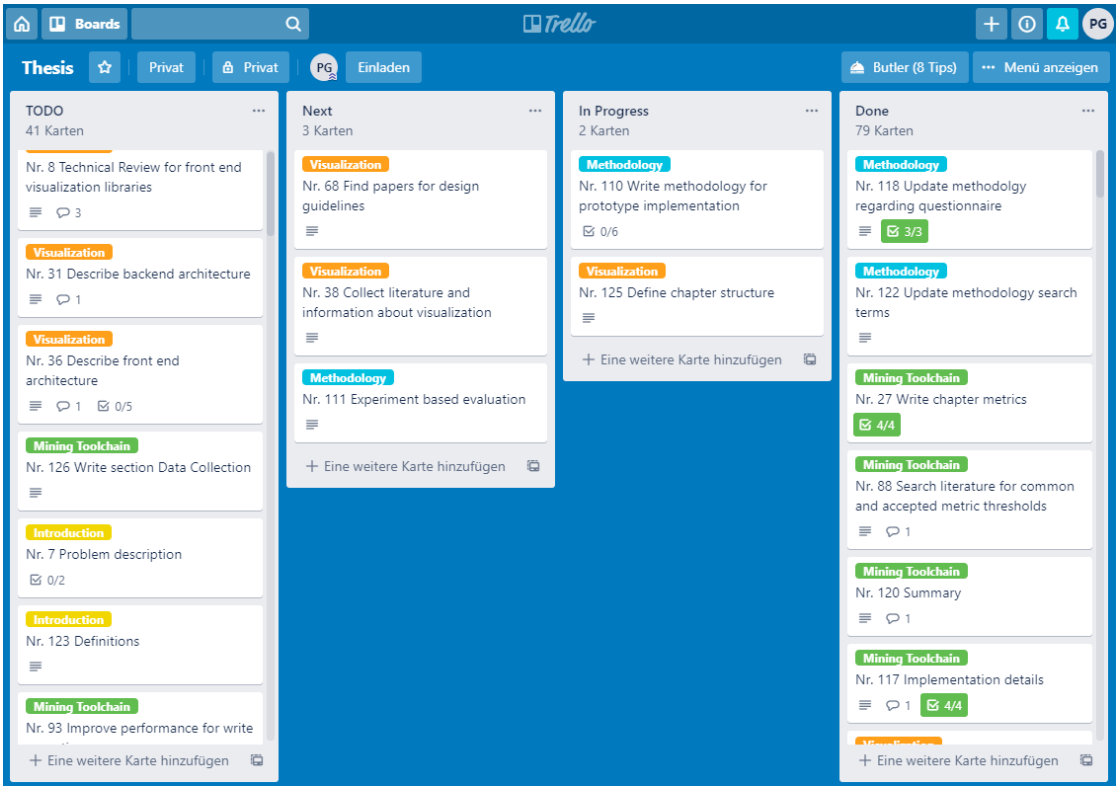


Figure 4.3: An online task board (hosted by Trello[7]) was used for organizing the work items

at this stage, or some information was still missing, the work item could also be postponed by moving it back to the *TODO* column.

3. **In Progress** Tasks in this column were actively worked on until all points in their specification were implemented correctly. If a task turned out to be too difficult to implement, or a subtask had to be completed first, the corresponding task could then either be postponed or broken down into further subtasks. These related tasks would then be added to the TODO or NEXT column, depending on the accuracy of their current specification. Working on several tasks simultaneously was possible, but in general no more than two or three tasks were in this state at the same time.

4. **Done** As soon as a work item was completed (all points in its *Definition of Done* list were finished), it was moved to this column and work on the next item could start.

[7] `www.trello.com`, last accessed on 23.02.2020

4.5.2 Development Tools

The following enumeration gives a brief overview of the tools used during the development process.

Source Code Versioning System All sources and documentation that were created during the development process were managed and versioned by using Git[8]. While Git supports different branching strategies that are very useful during collaborative work or in production environments, for this work, most of the time a single branching strategy was used where all changes were pushed directly into the master branch. If necessary, short lived feature branches were created and merged back into the master branch after completion.

Online Repository Hoster The Git repository used during the development process was hosted at the online provider Bitbucket[9]. This hosting environment provides an online service and web interface for storing private and public Git based repositories. The online repository was both used as backup store and also for synchronizing work between different work stations. For this thesis, a free, private repository was used.

IDEs Depending on the used programming language, different Integrated Development Environments (IDEs) were used. Python code, which was mainly used for the repository mining tool chain and for the visualization backend, was edited by using the free community edition of PyCharme[10]. Frontend related code, which was either in HTML or TypeScript, was developed with Visual Studio Code[11], a free source code editor based on the JavaScript Electron framework[12]. Editing other types of text files was mainly done by using Notepad++[13] or Textmate[14], depending on the used operating system.

Browser Testing the frontend components and interacting with the ArangoDB web interface was done with Google's Chrome Browser[15]. It should be noted that although the used web technologies should be supported by the latest versions of all currently available browsers, no additional compatibility checks were done during development.

[8] https://git-scm.com/, last accessed on 23.02.2020
[9] https://bitbucket.org/, last accessed on 23.02.2020
[10] https://www.jetbrains.com/pycharm/, last accessed on 23.02.2020
[11] https://code.visualstudio.com/, last accessed on 23.02.2020
[12] https://www.electronjs.org/, last accessed on 23.02.2020
[13] https://notepad-plus-plus.org/, last accessed on 23.02.2020
[14] https://macromates.com/, last accessed on 23.02.2020
[15] https://www.google.com/intl/de/chrome/, last accessed on 23.02.2020

4.5.3 Implementation process

Before starting the implementation tasks, a base line architecture for both applications (mining tool chain and expert visualization) was designed and improved continuously. Additionally, the following three models, among others, were created to describe the system's architecture:

- **Domain Model** An initial domain model [30] was created to collect and become familiar with the terms and concepts of the application domain. The required domain knowledge was gathered both through expert interviews and by systematic literature research.
- **Component View** In a next step, the different modules that comprise the overall system, and the interfaces they use for communication, were identified and visualized with the help of an UML component diagram [31].
- **Database Diagram** Saving the domain entities into a persistence store was done by the database layer. As this prototype uses a document-based graph database, instead of using a relational database scheme, the persistence model was expressed by a multi graph.

Moreover, some additional diagrams were used to describe specific aspects of the system, these will be listed separately in the respective chapters 5 and 6.

Implementation happened in small, iterative tasks by using recommended object oriented principles like *SOLID*[16] [32]. While development was not done in a strict test driven development (TDD) approach - which would mean that test cases would have been written prior to the actual implementation [33] - unit tests for most business logic were implemented as part of each task by stubbing out or mocking external dependencies [34]. More complex features that require interaction between different components or subsystems, on the other hand, were verified only by manual acceptance tests, since implementing automated integration- or UI-tests would have required too many resources and were out of the scope of this work.

4.6 Evaluation

After using experts interviews to validate the requirements, two different approaches were used to evaluate the two resulting systems of the prototype. The data mining tool chain was evaluated by a single-case mechanism experiment, while for the expert visualization, a scenario based expert evaluation was carried out.

[16] **S**ingle responsibility, **O**pen-close, **L**iskov substitution, **I**nterface segregation and **D**ependency inversion - are considered design principles to improve the maintainabilty of a software system.

4.6.1 Singe-Case Mechanism Experiment

A single-case mechanism experiment „is a test of a mechanism in a single object of study with a known architecture“ [24]. In case of the present data mining prototype, the *object of study* was a single software portfolio artificially constructed from selected source code repositories, which were chosen by a defined set of characteristics. The selection aimed at modeling a possible real world portfolio as good as possible. In turn, the so-called *mechanism to test* was the processing and analysis of this portfolio by the tool chain. Both, the execution times during the experiment and the generated analysis data were then evaluated against the expected results. The overall system performance was finally determined based on this comparison.

4.6.2 Scenario Based Expert Evaluation

For the final assessment of the visualization prototype, a scenario based expert evaluation was designed and carried out, similar to the *Technical Action Research* approach described in [24]. While evaluating the functionality of the system was also an aspect of this survey, the main goal was clearly to answer the previously defined research questions (see Section 1.3 in Chapter *Introduction*). During this process, a test plan with several real-world scenarios - based on verification and validation tests [35] - was designed, and a selected group of quality experts were asked to solve the given scenarios with the help of the expert system. Although the assessment of the system's usability was not the main focus of the survey, an additional system usability scale questionnaire [36] was also used to support the evaluation.

CHAPTER 5

Mining Toolchain

To visualize any quality metrics of a software portfolio, these must first be collected and prepared accordingly. Regarding the enormous size of today's software repositories, both open source and commercial ones, these data sets might span millions of code changes, split up into thousands of different revisions. To cope with these large amount of data interconnected data records, Robles therefore identifies two key factors that play an important role in collecting these data, *Data Mining* and *Automation* [38].

The following chapter gives an overview of how both concepts were incorporated into a data mining tool chain, which was used to gather all relevant data for the later visualization. Beginning with the definition of a workflow for mining source code repositories and specifying which requirements must be met by such an application, the implementation and validation process that lead to the resulting tool chain will then be described in detail.

5.1 Data Mining Process

Most publications in this area use a three or four phase model to describe a data mining process for mining source code repositories [11, 37, 38, 40, 42]. While these individual phases can have different names depending on the publication, they more or less each have the same purpose and are also executed in the same, successive order:

1. **Data Collection** During this phase, the relevant information must be extracted from the source code, usually by first creating a local copy of the repository. Depending on the used version control system, different APIs must be accessed for this task, which does not only include the retrieval of the source code itself but also its meta data information (author and date of the commit, but also a textual description of the changes) [42]

2. **Measurement and Analysis** This phase is responsible for calculating various metrics, and in some cases, also for some analysis tasks. Depending on the applied workflow, this can either take place before [37, 38] or after [40] storing the collected meta data in a persistent storage. While for some metrics, static code analysis may be sufficient, other metrics might require compiling and executing the sources beforehand (e.g. when determining the test coverage). However, creating and executing binaries from code can sometimes be very time-consuming [37] and might even require additional manual steps, such as resolving missing dependencies [11].

3. **Storage** In this phase, both meta data of the collected artifacts and the associated metrics are stored in a persistent storage like a relational database system [37, 38, 40] for later analysis by the user.

4. **Reporting, Visualization** Analysis and visualization of the collected data is also considered as part of the data mining process [38, 42], but since the visualization is a major part of this work, this step is discussed in more detail in a separate chapter.

Automation plays another critical role during this process, since collecting data and calculating metrics manually are very tedious and error-prone tasks and are therefore very impractical for most research projects [37]. This is even more the case for commercial areas, as here quality managers often have a number of additional tasks to accomplish during their daily work [17].

Figure 5.1 shows a schematic view of the data mining process that was realized during this research, the aforementioned four phases are organized from left to right: During phase one, the repositories of the portfolio are downloaded and cloned into local repositories. After that, an external metric calculation tool calculates the metrics for the corresponding source code and stores them in a local, file-based database (see section 5.3) from where the metrics will are loaded into an in-memory data structure for further processing. This process has to be repeated for every revision of the repository, or at least for every revision that should be analyzed. The idea behind this post processing is to reduce the computation time during the visualization phase, as the more pre-calculated data can be retrieved directly from the database, the less has to be calculated during runtime. The post processed data is then be merged with additional meta data and stored in the database during phase three. Finally, the user frontend accesses the database through a backend application and executes queries on it (phase four, see Chapter 6).

5.2 Requirement Specification

The requirements for the data mining process were defined based on a systematic literature search regarding existing research papers on source code repository mining. Some studies already defined concrete requirements as part of their research [42], while other papers specifically identified challenges in developing such a process [11]. Their findings could also be used to derive possible requirements.

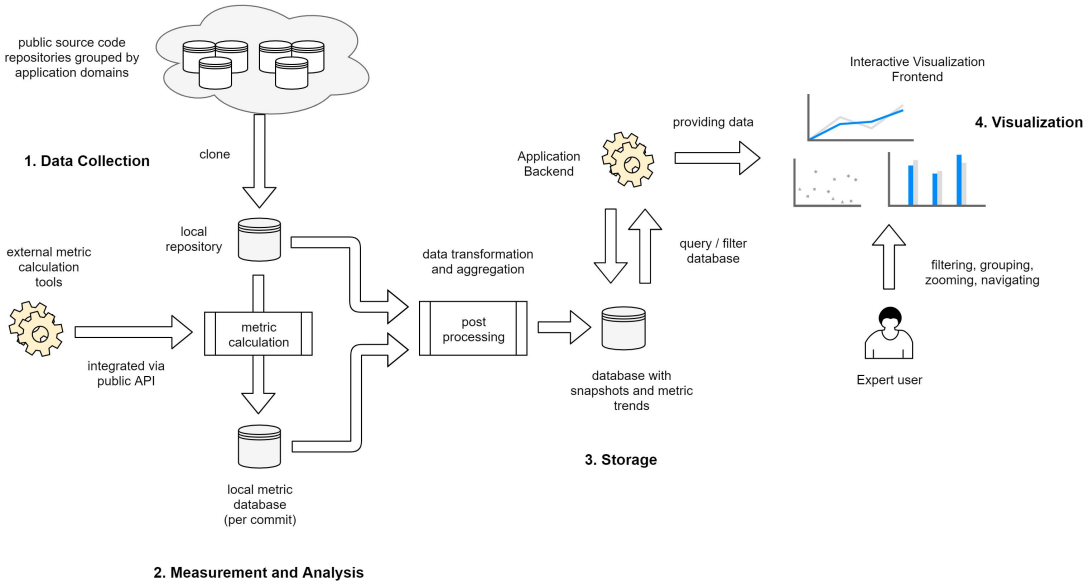


Figure 5.1: A schematic view of the data mining workflow

Since the data mining process, in contrast to the visualization component, should primarily run as an automated background task without any, or at least with only minimal, direct user intervention, additional expert interviews to elicit any user interface related requirements were not carried out.

The following list contains a summary of the requirements that have been identified during the research and were taken into account during the development process:

M.R1: Modularity and Extensibility
This prototype should ideally be a starting point for further, ongoing development, which also means that its requirements might change in the future, especially regarding the types of metrics or repositories that should be supported. To take this into account, the system should provide a modular architecture that could be extended without requiring larger rewrites of the core system [38].

M.R2: Integration of external sub systems
Calculating metrics can be very complex and therefore should ideally be factored out to external tools [38]. However, it must be taken into account here that different tools can often calculate different metric values for the same artifact [14] and therefore it could be necessary to switch out the metric calculation for getting more accurate results. Integrating external metric tools in the tool chain should therefore be possible without too much configuration. The same also goes for accessing source code repositories. While for the current work, it might be sufficient to support only one source code repository, it could become necessary in the future to support different types of source code version control systems.

M.R3: Automated batch processing
Downloading and collecting metrics from different repositories could take up to several hours for larger projects, it would therefore be good if the whole process could be started for the whole portfolio at once and could then run unsupervised until finished, for example over night. This requires the tool chain to be able run mostly automated after the user has defined some initial start-up parameters. Additionally, this prototype was limited to metrics only available through static code analysis, because gathering runtime metrics would require code compilation which sometimes need manual interaction, as some source code repositories can have missing dependencies or syntax errors that need to be fixed before [11].

M.R4: Fast computation
Although the workflow should ideally run fully automated, this does not mean that it should take unnecessary long. While it might be difficult to define any concrete numbers here for what might be a reasonable processing time, an empirical value that has been shown to be suitable during development would be approx. 15 to 30 minutes for the complete analysis of an average project (50,000 - 100,000 LOC, 50 Commits to analyze). To improve processing time and resource usage, bulk database write operations and asynchronous processing [37] could be used (see section 5.4). Also, external metric providers should be evaluated based on their performance, which will be discussed in section 5.3.

M.R5: Fault tolerance
Many different inputs form external sources will be processed during data mining and it cannot be guaranteed that this input data is always well formed. Source code can have syntax errors [11] or could use language features which are not (yet) supported by external metric calculation tools (see section 5.3). To handle these cases, the mining tool chain must be implemented with a certain level of fault tolerance in mind so that it can still operate even if one of its modules terminates unexpectedly.

M.R6: OOP language support
Due to limitations regarding time and development resources, this prototype focuses mainly on object-oriented metrics [22] (while other metrics could still be added later), and therefore the analysis should also be targeted at object-oriented languages.

One problem here is that object-oriented languages are not homogeneous and can use different mechanisms for encapsulation or inheritance, which can result in different values for specific metrics. While languages like C++, Java or C# use similar mechanisms like static inheritance, modifier-based encapsulation and others, there are also languages like JavaScript which use different approaches like prototyped-based inheritance. This circumstance, for instance, could affect some metrics like the depth of the inheritance tree or the number of subclasses. Additionally, prototype based inheritance structures can be altered during runtime, which makes it difficult for static analysis tools to gather all relevant information.

While JavaScript does support class-based inheritance since the ECMA Standard 6[1] (at least on syntactical level), it cannot be assumed that this language feature is used by all JavaScript projects within a portfolio.

A study [43] has shown that the differences in metrics between statically typed, inheritance based languages like C++, Java and C# are not that significant, so comparing projects written in these languages could lead to reasonable results. Therefore, the mining prototype, and the external metric tool used by it, currently focus on analyzing Java and C# repositories.

M.R7: Commonly used architectural standards
Since the present work is a prototype which may be followed by further development, there should be a low entry barrier for new developers joining the project. To achieve this, the architecture of the mining tool chain should use common software concepts like design patterns [44] and should also be based on an extensible framework design [45]. Also, a common, suitable programming language and technology stack should be used for the implementation to help other developers getting started more easily. Details regarding the architecture and implementation will be discussed in section 5.4.

The remaining sections of this chapter describe in detail how these requirements were realized in the mining tool chain and what effects they had on implementation and design decisions.

5.3 Metric Providers

To save development time, the required metrics should already be calculated by third-party applications, and therefore various external metric tools were evaluated during this study. Lincke, Lundberg, and Löwe [14] contain an overview and evaluation of some selected software metric tools. In addition to these suggestions, other applications were collected using an internet search and finally selected by using the following criteria (the corresponding requirements, as defined in 5.2, are added in brackets):

- Is the tool freely available and does its license allows free usage for research purposes?
- Can the tool be easily integrated into the mining tool chain (*M.R2*)?
- Does the metric calculation take a reasonable amount of time (as this directly affects the overall runtime, see *M.R4*)?
- Can the tool handle syntax errors or missing references in code (*M.R5*)?
- Does the tool work with actual language versions of Java or C# (*M.R6*)?

[1] `http://www.ecma-international.org/ecma-262/6.0`, last accessed on 30.01.2020

5.3.1 Tool Selection

The metric tools which have been shortlisted for integration into the mining tool chain were the following:

ckjm — Chidamber and Kemerer Java Metrics

Developed by Diomidis Spinellis [46], *cjkm* is the first in this list. It is a very lean tool that provides only a Unix-like command line interface which would make it a good candidate for being called as an external process during the data mining process. It supports calculation of some of the most common object-oriented metrics [22] for the Java language. *ckjm* is freely available under the Apache 2.0 license[2] . As it operates on the class files of a Java projects, compilation of the sources is required up front.

cccc - C and C++ Code Counter

Despite its name, this source code analysis tool also supports Java sources. It was developed by Steve Arnold and is freely available under GPL2 license[3], but at the moment is not under active development anymore. Similar to *ckjm* the tool provides a command line interface and calculates a number of metrics like lines of code, Fan-In, Fan-Out or cyclomatic complexity (see section 2.2 for a description of the different metrics). What makes the use of *cccc* a bit problematic is the fact that it only generates HTML reports as output. While parsing HTML files to get the relevant information would be possible, a cleaner, maybe pure text based, output format would be more feasible for integrating the tool as external process.

SourceMeter

SourceMeter is a commercial tool developed by a company called FrontEndART [4] with free versions[5] for different languages like Java, C# or C++ available. It does support metric calculations and also statically checks for coding rule violations. Its command line interface allows for many different configurations and it also supports a rich set of different metrics for different areas like complexity, coupling, documentation and others. Unfortunately, *SourceMeter* does only support C# projects up to version 6.0[6], which, at the time of this writing, is already five years ago. How this affected the analysis will be discussed in section 5.3.2.

[2] `https://github.com/dspinellis/ckjm`, last accessed on 30.01.2020
[3] `https://github.com/sarnold/cccc`, last accessed on 30.01.2020
[4] `https://www.sourcemeter.com/`, last accessed on 30.01.2020
[5] `https://www.sourcemeter.com/terms-of-use/`, last accessed on 30.01.2020
[6] `https://www.sourcemeter.com/features/`, last accessed on 30.01.2020

MSBuild

By installing an additional Nuget package, *MSBuild*, the build tool of Microsoft's Visual Studio, also supports the calculation of some metrics via command line[7]. The calculated metrics are stored in an XML file and could therefore be easily accessed by an external process. MSBuild generates only a few metrics like lines of code or complexity. Compared to other tools like *SourceMeter* or *cjkm*, its scope is a bit limited, also metrics can only be calculated for C# projects. The latest version of MSBuild is published under an MIT license[8].

SourceMonitor

This is a tool developed by Campwood Software[9] and provides both a graphical user and a command line interface. As most of the other tools in this list, it is freely available and does also support languages like Java and C#, among others. It mainly supports some standard metrics like counting lines of code or numbers of classes or interfaces within a project, but does not support more complex object-oriented metrics as the ones defined in [22].

Understand

Understand from Scitools[10] is a commercial static code analyzer but is freely available for research use under a Creative Common License[11]. While it has a graphical user interface, it can also operate via command line and provides a batch mode, where several commands can be combined and executed via a single call to the Understand process. It supports calculating various complexity, volume and object-oriented metrics for different languages, including Java and C#. Some newer features, like expression bodies for properties introduced in C# 7.0[12], are not supported.

5.3.2 Evaluation

Evaluation of the different tools was mostly done by consulting the documentation provided by the tool developers. Where sufficient enough, the online documentation was used and in case it was either not available or not detailed enough, the tools were installed and the user manual, if available, was then used for further research (which, for

[7] https://docs.microsoft.com/en-us/visualstudio/code-quality/how-to-generate-code-metrics-data?view=vs-2019#command-line-code-metrics, last accessed on 30.01.2020

[8] https://github.com/microsoft/msbuild/blob/master/LICENSE, last accessed 30.01.2020

[9] http://www.campwoodsw.com/sourcemonitor.html, last accessed on 30.01.2020

[10] https://scitools.com/, last accessed on 30.01.2020

[11] https://scitools.com/student/, last accessed 30.01.2020

[12] https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7, last accessed on 30.01.2020

instance, was the case with *SourceMeter*). License information was in all cases available through the developer's website.

A pre-selection of the tools was done based on the following criteria: *Interface* (how easily can the tool be integrated), *supported languages* (especially, Java and C# were the preferred languages here - see *M.R6* for a more detailed explanation), *number and type of supported metrics* and *license regulations*.

Table 5.1 compares the different code metric tools based on the aforementioned criteria. For each criterion, the number of points achieved is given in the row below (one star = solution meets the requirement only on a very low level or not at all, three stars = satisfies requirement completely).

Criteria/ Tool	ckjm	ccc	SourceMeter	MSBuild	SourceMonitor	Understand
Interface	cmd line	cmd line, HTML out	cmd line	cmd line	cmd line	cmd line and batch mode
Score	★★	★	★★	★★	★★	★★★
Supported Language	Java (class files)	C, C++, Java	C++, Java, C# (6.0)	C#	C, C++, Java, C#	C++, Java, C# (7.0)
Score	★	★★	★★	★	★★★	★★★
Metrics	complexity, OOP	complexity, some OOP	complexity, OOP, coupling, doc	complexity, OOP, few coupling	complexity	complexity, OOP, volume
Score	★	★	★★★	★	★	★★
License	Apache 2.0	GPL 2	free for non commercial use	MIT	Freeware	Creative Common for educational use
Total Score	★	★	★★	★	★	★★★

Table 5.1: Comparison of different metric providers

All tools had a command line interface that allowed for easy integration into the mining tool chain. The fact that *cccc* uses HTML as output format led to point deduction, because this would require an additional parsing step to get the results. Due to its batch mode and extensive configuration settings, *Und*, the command line interface of *Understand*, could gain the most points in this area.

Listing 5.1 shows an example batch file for *Understand*. The file creates an Understand database for the given repository (1), configures the metrics that should be calculated (3) and starts the two pass calculation process of Understand (5, 6).

```
create -languages C# ./metrics/NUnit/_db
add ./source/NUnit
settings -metrics CountLineCode RatioCommentToCode
settings -MetricsFileNameDisplayMode RelativePath
analyze
metrics
```

Listing 5.1: Understand batch support

Considering supported languages and language versions, *Understand* was again ahead, since it supports both Java and a newer C# version than comparable tools like *SourceMeter*, and was also able to ignore syntax errors in the code.

Even if the mining tool chain accesses the metric tools via a generic interface, implementing this interface per external tool requires a certain effort. Therefore, considering implementation time, the more metrics are supported by a single tool, the better. *SourceMeter* offers probably the largest amount of available metrics, in total it supports six different areas of metrics (although, assignment of a metric to a specific category is not always unambiguous among all tools). *Understand* offers a few metrics less here, but still far more than the other programs which were part of this study.

Regarding license regulations, all tools provided a free version, at least for educational use, so this criterion was not explicitly evaluated.

Based on a pre-evaluation of these criteria, two candidates remained, *SourceMeter* and *Understand*, which were further investigated by an experimental study to verify their fault tolerance and performance. For this, two exemplary projects were chosen for which their metrics should be calculated by the tools.

The one project was NUnit[13], a unit testing framework developed in C#, and the other one was JUnit5[14], its Java based equivalent. Both repositories have about 5000 to 6000 commits each, but Junit5 might have a slightly richer function scope. While the Java analyzer of *SourceMeter* is able to process metric calculation by simply providing a base directory, the C# analyzer requires either a solution file, a project file or a single source file (using wildcards in the file name was not possible).

Analyzing of the JUnit5 source code repository took *SourceMeter* about three minutes and *Understand* approx. two minutes, both tools generated reports in csv format.

Calculating metrics for NUnit with *SourceMeter* threw an error during its *SolutionAnalysisTask* due to an unsupported solution/project format and the analysis was aborted by the tool. Analyzing NUnit with *Understand* took less than a minute, the fact that NUnit is a smaller project compared to JUnit5 might well have played a role here. Although NUnit uses some newer C# language features that *Understand* could not validate during analysis, it was still able to finish the metrics calculation.

While regarding the runtime performance, both final candidates were almost on par, *Understand* ultimately offers better language support and its command line mode is more powerful because of its batch file support. The decision was therefore made in favor of *Understand*.

5.4 Implementation

The implementation of the data mining tool chain was performed in an iterative process. Based on the requirements defined in section 5.2, corresponding tasks were created and placed on an online task board[15].

[13] `https://github.com/nunit/nunit`, last accessed on 01.02.2020
[14] `https://github.com/junit-team/junit5`, last accessed on 01.02.2020
[15] `https://trello.com/`, last accessed on 02.02.2020

While some initial tasks were self-contained on their respective technology or infrastructure layer and therefore had to be completed upfront, like defining an architectural base line and setting up the fundamental database structure, most other tasks were executed in the form of vertical slices [52]. Using this approach, implementation of a single feature was done through all layers until the desired requirements were met. In that way, the implementation could focus on the current, specific problem, instead of defining the whole system upfront without knowing possible challenges that might arise during development.

The following section first gives an overview of the used technology stack before describing the system and database architecture that was built upon this stack. The section concludes with some implementation details and challenges that are worth mentioning.

5.4.1 Technology Stack

To choose an appropriate technology stack, first all external subsystems to which the system had to connect to had to be identified. This also affected the programming language that was finally selected for the prototype, since it had to be checked how communication with the subsystems could be established (via REST interface, native library etc.).

Another elementary decision was to chose a suitable database for storing the relevant metrics and meta data. Based on the positive experiences from former projects and master theses performed at the institute, the choice fell to **ArangoDB**[16], a document-based NoSQL graph data base. Thanks to its document-based approach, it allowed for easier prototyping, as the different document schemes did not have to be defined upfront, but instead could be refined during development. Furthermore, its ability to link documents together in a graph-like manner simplified the definition and querying of relations between document entities. ArangoDB has its own query language called ArangoDB Query Language (AQL). The language is quite similar to SQL but provides additional features for document or graph traversal operations. A detailed description of the underlying database model is described in section 5.4.3. Programming libraries for interacting with an ArangoDB database are available in most mainstream programming languages.

On the top-most system layer, the tool chain had to communicate with external source code repositories to download repositories and gather revision meta data. While there are various different source code providers and technologies available, many of them use the source code version system Git[17] as their foundation. This prototype therefore also focused on accessing public **GitHub**[18] repositories. GitHub is an online service specialized on hosting public and private source code repositories based on Git. While in the past, many studies dealing with source code repository mining used SourceForge[19]

[16] `https://www.arangodb.com/`, last accessed on 02.02.2020
[17] `https://git-scm.com/`, last accessed on 02.02.2020
[18] `https://github.com/`, last accessed on 02.02.2020
[19] `https://sourceforge.net/`, last accessed on 02.02.2020

as their source for public repositories, in the meantime GitHub became the number one choice for source code repository hosting [53]. Repositories and their metadata can be accessed via a public REST API and there are also different programming libraries available that are based on this API.

It should be noted in that context that, due to requirement *M.R1*, the supported repository technology might be subject to change and therefore GitHub was only one possible candidate, which was, because of its easy integration and large amount of available repositories, used during the prototype development.

Python[20] was finally chosen as programming language for implementing the modules of which the mining tool chain was composed of. Python's dynamic typing and multi-paradigm approach makes it a good choice for fast prototype development and its large ecosystem provides many external libraries for all kind of different problems. Among these libraries are also some for data intensive calculations, but thanks to the operator functions provided by ArangoDB, many of these calculations could be outsourced to the database server. For accessing the GitHub repositories with Python, the library *PyGithub*[21] was used. Data mining the downloaded repositories was done with the help of *PyDriller* [54], a library especially for mining Git repositories. By using the integrated Python package Lizard[22] PyDriller provides also some basic data mining capabilities and source metric calculations, but currently only a few complexity and structural metrics are available. For reading and writing from and to the ArangoDB, the Python library *PyArango*[23] was used.

5.4.2 System Architecture

The following system overview starts from a bird's eye perspective showing the modules and their interaction and consecutively drills down into a more detailed view of the domain model and its realization within the architecture.

Modules

The system's component design (illustrated in Figure 5.2) was chosen to be similar to the one suggested by [38]. The main process in the middle of the diagram is the equivalent of the core module mentioned by Robles [38], and as such it is responsible for orchestrating the workflow between the different components. Therefore, it first reads all relevant user defined configuration data (like database connection data, repository identifiers which are part of a portfolio etc.) and then starts the mining process by requesting the repositories from the download module. In the next step, the metric analysis starts for every snapshot of a repository and the results are fetched by the main process for storing them in the database, together with additional meta data from the snapshots. In a final

[20] `https://www.python.org/`
[21] `https://pygithub.readthedocs.io/en/latest/`, last accessed on 02.02.2020
[22] `https://github.com/terryyin/lizard`, last accessed on 06.02.2020
[23] `https://pyarango.readthedocs.io/en/latest/`

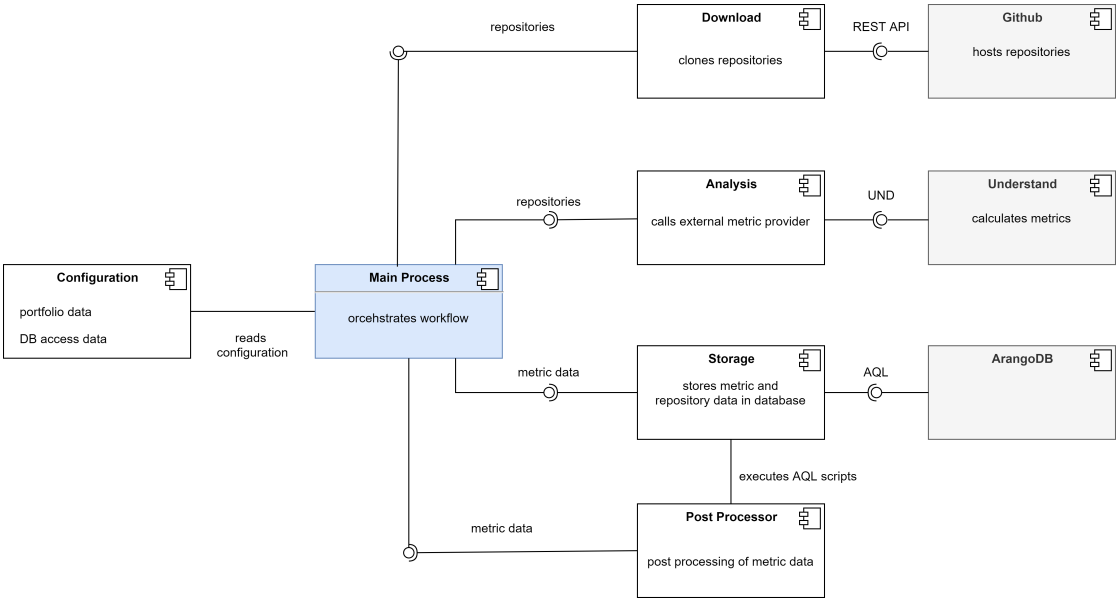


Figure 5.2: UML 2 Component view of the mining tool chain

step, the post processor generates additional aggregated data of the metrics and stores them directly in the database.

To reduce coupling within the system, components generally communicate with each other only via the main process, the only exception being the post processor, which sends its queries directly to the storage module. This is because these queries are usually plain AQL scripts which the ArangoDB driver can immediately execute, and any additional routing over the main process would make the system only unnecessarily complex.

Domain Model

The domain model in Figure 5.3 provides a more detailed view of the concepts that were relevant during the development, external subsystems or access to them is distinguished by using a different color encoding (red for persistent storage, blue for external metric calculators and green for remote repository access).

Formalizing these concepts in the domain diagram has mainly two purposes here: (1) It should help in defining an ubiquitous language so that all concepts that are used throughout the system can uniquely be identified by its name and (2) it ensures that certain domain concepts are made explicit which would otherwise be hidden [30].

While the ubiquitous language is in the first place more useful for improving communication between developers and stakeholders in larger teams, it can also be helpful for identifying or becoming aware of all relevant domain concepts.

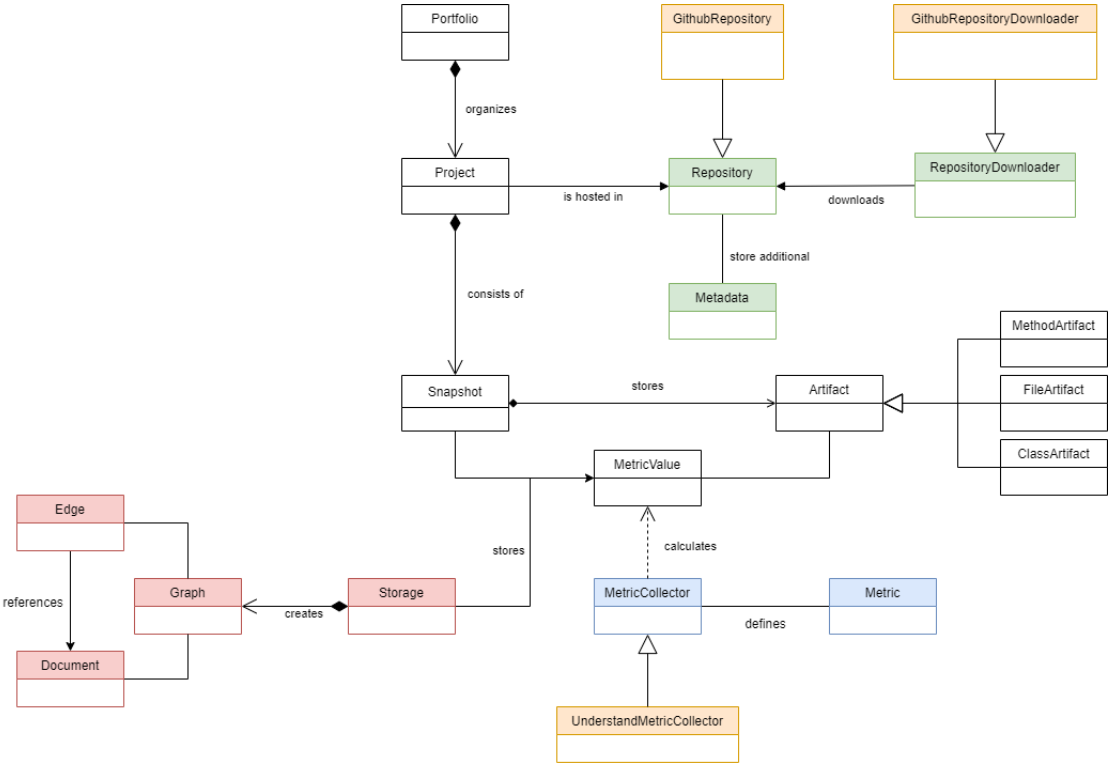


Figure 5.3: Domain model of the mining tool chain. Red elements represent concepts related to the persistent storage, interfaces to external components and the data they deliver are either blue (metric calculators) or green (repository access). Variable points that implement access to external components are painted in yellow.

Most of the domain concepts relevant for the mining process are identical to the ones described in more detail in Chapter 2.

Extendibility

To satisfy requirement *M.R1* and provide a certain level of extendibility, the concept of hot spots in a white box framework [45] were used in the application. These hotspots define generic interfaces within the system that have to be implemented for more concrete use cases.
In that way, the general workflow is still predetermined by the framework, but specific application scenarios can be realized by exchanging the variable parts of the system. This allows for extending the system without changing its existing components, a concept specified as *Open/Close Principle* [56].

One of these spots specifies the connection point to remote repositories. Based on [42], accessing a remote repository for data mining requires at least three operations, *checkout*, *update* and *log*. The *checkout* operation should download the repository from its remote

source and clone it into a local repository. The *update* operation on the other hand is responsible for retrieving a specific revision of the repository and the *log* operation can finally be called to retrieve additional meta data of the current revision.

This nomenclature reveals a bit of a naming conflict, as in the context of git repositories, the term *checkout* is normally used for switching branches, while in this repository-agnostic terminology it was used for downloading a repository (which in git terms is called *cloning*).
The present implementation uses Python's concept of an abstract base class[24] called RepositoryBase (Listing 5.2), to define this interface.

```
class RepositoryBase(abc.ABC):
    """
    should be implemented by derived classes
    to access specific repository types
    """

    @abc.abstractmethod
    def checkout(self, download_folder):
    """should download the remote repository to the given folder."""
    pass

    @abc.abstractmethod
    def update(self, snapshot):
    """updates the local repository to the given snapshot"""
    pass
```

Listing 5.2: Abstract base class for repository interface

Also note that the *log* method is omitted in this interface, as the snapshot's meta data are already provided by an instance of the `RepositorySnapshot` class that is used for updating the repository.

A specific implementation of this base class is provided by the `GithubRepository` class which handles the concrete use cases of downloading and accessing GitHub repositories. Since the repository instance are created by the download module and not the main module, an additional factory class [44] is used to delegate the instantiation of the repositories. In that way, configurations of the different hot spots can be kept isolated and do not have to be scattered over different modules.

A similar approach was also used to make the metric collector sub system interchangeable. For supporting a different metric collector, all that has to be done is to provide a new subclass of the `MetricCollectorBase` class.

5.4.3 Database Design

Thanks to ArangoDB's document-based architecture, the database design could be developed iteratively and the data model could be successively enriched with additional data. Within an ArangoDB database, data is organized in the form of inhomogeneous document collections,

[24] `https://docs.python.org/3/library/abc.html`, last accessed on 03.02.2020

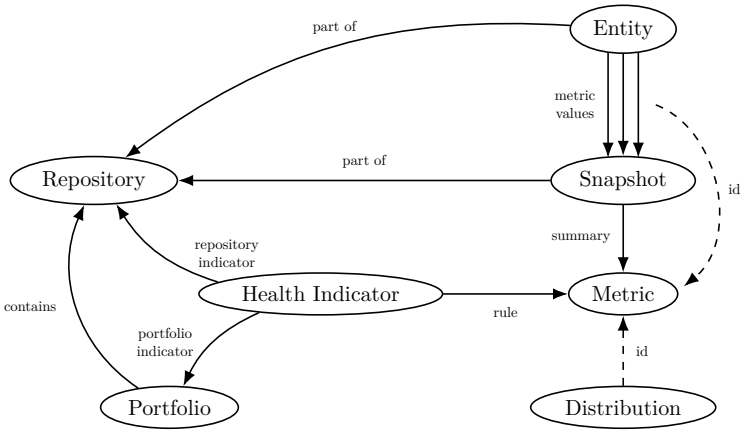


Figure 5.4: Graph structure of ArangoDB database model

whereby each collection can contain a large number of schema free documents. This structure is similar to most document-based NoSQL databases like MongoDB[25] or CouchDB[26].

Moreover, ArangoDB also supports graph like structures by interpreting documents either as nodes or, in the case of relations, as edges.
By traversing this graph structure, one can efficiently navigate through document relations without having to execute performance-intensive join operations, as it would be necessary with relational databases.

Figure 5.4 illustrates the implemented database design in the form of a an directed multigraph. It should be noted that, although the graph is directed, by using AQL, traversing in both directions of an edge is possible.

Most domain concepts like *Portfolio*, *Repository*, *Snapshot* and others have a direct representation as document nodes and their additional meta data is stored as key-value tuples within their documents. Beside these isolated concepts, there are also some domain concepts which are related to more than one entity and therefore were modeled as edges between these document nodes. This was especially the case for the following concepts:

Metric Values An artifact's metric value for a snapshot is modeled as edge between its *Artifact* and *Snapshot* document. The calculated value is thereby a property of the edge itself. Since multiple metrics can be calculated for one artifact, its node can consequently also have multiple connections to a snapshot node, each of these edges representing a different metric value.

Summaries and aggregated metric values are calculated during the post processing phase to improve the system's reaction time during the interactive analysis phase. These summed up values could be, for instance, the overall source lines of codes for a whole project or the average depth of the inheritance trees per class. The aggregated value of any metric is calculated on a per snapshot basis, so it makes sense to model this domain concept as an edge between the *Snapshot* and the corresponding *Metric* node, where the aggregated value is again a property of the edge itself.

[25] `https://www.mongodb.com/`, last accessed on 04.02.2020
[26] `https://couchdb.apache.org/`, last accessed on 04.02.2020

Health Indicators are a common concept borrowed from existing quality tools like SonarQube. Their main purpose is to indicate the current portfolio state based on certain criteria. This concept becomes more relevant during the visualization phase and is only mentioned here for the sake of completeness, but does not have any relevant impact on the mining process itself.

Modeling some entities as edges allows for easier search queries within the graph, for instance, to retrieve the summarized line-of-code metrics for all snapshots of a repository, an AQL query like the one in Listing 5.3 could be used. Starting at the repository with the given key (this example uses only a place holder) the query traverses over the incoming edge (marked by the `INBOUND` keyword) to the *Snapshot* documents that are connected with this repository. From there, it continues with traversing all outgoing (note the `OUTBOUND` keyword) edges to the line-of-code *Metric* document. The metric value itself is stored inside the traversed edge and can be accessed through the `_value` property. The query finally constructs a JSON result set from the relevant nodes and edges.

```
FOR r IN Repository FILTER r._key == 'REPOSITORY-ID'
FOR s IN 1..1 INBOUND r._id SnapshotOfRepository
FOR metric, metric_value IN 1..1 OUTBOUND s._id SnapshotSummary
FILTER metric._key == 'CountLineCode'
RETURN {
    metric: metric._key,
    snapshot_id: s._key,
    metric_value: metric_value._value
}
```

Listing 5.3: AQL query for retrieving metric values

5.4.4 Implementation Details

The following section presents some aspects of the implementation which are worth mentioning. This includes both performance optimization regarding the analysis and data storage phase, but also some situations where the implementation required some unexpected workarounds.

Configuration

While most compiled applications use external configuration files in, for instance, JSON- or XML format, and load these files during runtime, the fact that Python is an interpreted language opens up the possibility of also storing the entire configuration directly in Python files.

While this does not allow changing the configuration during runtime (which, for this work, was not a requirement of the mining tool chain prototype), it is still possible to edit the configuration and restart the application without recompiling. Furthermore, using a fully general-purpose language here made the configuration much more powerful than it had possibly be with other file formats - although it must be noted that due to this decision, some basic Python programming skills are required for configuring the system.

The configuration itself was organized in a package consisting of different modules, each one responsible for an isolated configuration aspect, like database credentials, portfolio composition or GitHub connection data.

Parallelism

The repository mining tool chain has to execute several long running tasks, like downloading sources, running metric analysis on selected snapshots, calculating aggregations and distributions and finally storing all the gathered data in the database. While some of the operations require sequential processing, others can be executed in isolation from each other, which makes them good candidates for parallelism. The idea behind parallel processing is to execute different tasks simultaneously to better use existing system resources which would otherwise be idle [57].

Python's *multiprocessing*[27] package offers various ways of parallel programming, of which the `Pool` class is best suited for data parallelism. When applying data parallelism, a set of input data is divided among different worker jobs organized by a process pool. These workers are then running simultaneously and the results of each job are finally merged together into a single result set.

Before applying parallelism, the tasks which can be executed in isolation with a minimum of interprocess communication or synchronization have to be identified first [57]. Regarding the repository mining process, the following tasks could be subject for parallelism:

1. Downloading the repositories from a remote provider would be a good candidate, as the download process of every repository happens in isolation and no communication is required between the different download processes.
2. Analyzing the commits of a single repository in parallel is problematic, as the metric calculator depends on the current state of the repository on disk and checking out a specific revision would change this state. The only solution here would be to checkout any revision to a different location on disk, but this would be very impractical due to the large amount of disc spaces which would hereby be required.
3. Analyzing several repositories simultaneously on the other hand would work much better. Here, the repository's physical location would only be accessed by one process at a time and the different revisions would be checked out sequentially within the same process.

For the current prototype implementation, only the first task from the above list was parallelized, while the third point might be an interesting option for possible future releases.

Performance Optimization

As the previous section has illustrated, while some performance improvements could already be made by simultaneously downloading the repositories, there are still some other possibilities to improve the execution time of the mining process. But since implementing these optimizations can be a complicated, resource-intensive task, any optimization should be carefully considered, as already mentioned by the frequently quoted sentence from Knuth: „Premature optimization is the root of all evil (or at least most of it) in programming“ [58].

To avoid unnecessary optimization, first the current execution time of different aspects of the mining process was monitored, and after that, the parts where optimization could lead to better performance, where identified. Finally, the execution times were measured again to be able to assess the adjustments.

[27] `https://docs.python.org/3/library/multiprocessing.html`, last accessed on 08.02.2020

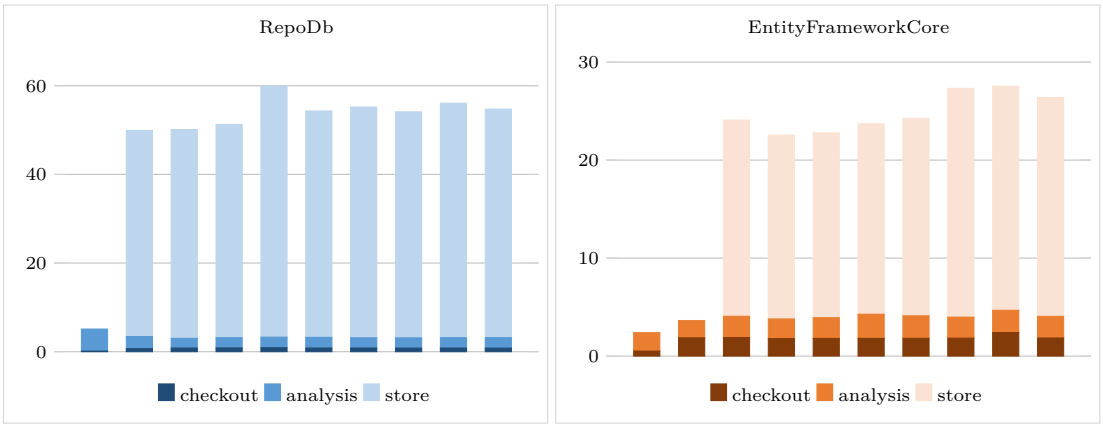


Figure 5.5: Execution time of different mining phases (in sec) for the first ten commits of RepoDB (left) and Entity Framework Core (right)

To evaluate the performance, two repositories - RepoDB[28] and EntityFrameworkCore[29] - were downloaded from GitHub and the mining tool chain was used to analyze the first ten commits of each of them. The chosen repositories differ in size and number of commits and therefore provide a good sample regarding various project sizes. Additionally, both are actively developed, which allows the presumption that at least some of the newer C# language features are used. The results are illustrated in Figure 5.5: Every entry on the x-axis represents a commit of one of the two repositories and the y-axis show the execution time in seconds for the different mining phases per commit.

These benchmarks revealed some interesting details: First, the lack of writing time for the first two commits might have been due to the fact that there were no files yet to be analyzed and written (very often, initial repository commits consist only of a few files like readme, gitignore etc.). Furthermore, it seems like that the analysis phase might have required an almost constant amount of time, no matter how many files there had to be analyzed. This might be because of some initial startup overhead when calling the external *Understand* process. But what catches one's eye immediately is the fact that the time for downloading and analysis was almost negligible compared to the time required for storing all relevant data into the database. So any possible performance optimization had to focus on this phase.

By investigating one of the different operations for writing domain objects into the database (see Listing 5.4), some possible performance issues could be identified: (1) Before writing a domain object to the database, it has to be checked if this object already exists (line 4) and (2) for every domain object, a single write operation - the call to `createVertex` (line 8) - is performed.

This resulted in at least two separate database access calls for any domain object. While this might not be such a big problem when storing snapshot data (the number of snapshots to be analyzed can be configured), it can become critical regarding the amount of artifacts or metric values per snapshot, as these could easily sum up to several thousands of records that have to be stored in the database.

[28] `https://github.com/mikependon/RepoDb`, last accessed on 09.02.2020
[29] `https://github.com/dotnet/efcore`, last accessed on 09.02.2020

```
def get_or_create_snapshot(self, snapshot_id: str, snapshot_date: datetime)
    :
   try:
     # check if snapshot already exists in DB
     snapshot = self.database[Snapshot.__name__][snapshot_id]
     return snapshot
     except DocumentNotFoundError:
       # create new snapshot and store it in DB
       commit = self.graph.createVertex(Snapshot.__name__,{
              "_key": snapshot_id,
          "date": snapshot_date
       })
   return commit
```

Listing 5.4: Saving a single snapshot to the database

To improve the performance of the storage phase, the roundtrips to the database had to be reduced to a minimum. This was achieved by using bulk write operations which are provided both by ArangoDB and its Python interface[30]. During a bulk save, multiple documents are written at once to the database, which can improve performance significantly, as the connection overhead required to access the database is reduced to a single call. Listing 5.5 shows a possible implementation: First, a list of relevant snapshot meta data is created and then the complete list is stored at once (line 11). Since all snapshots are stored at once, there is no need for checking whether a snapshot does already exist in the database (assuming the repository is analyzed for the first time). This approach requires only two database accesses, the first to retrieve a reference to the collection and the second one to store all documents into this collection.

```
def store_snapshots(self, snapshots: List[RepositorySnapshot]):
    snapshots_to_add = []
    # create an in-memory document per snapshot object
    for snapshot in snapshots:
        snapshots_to_add.append({
          '_key': snapshot.key,
          'date': snapshot.date
        })
    snapshot_collection = self.database[Snapshot.__name__]
    # write all documents at once to the DB
    snapshot_collection.bulkSave(snapshots_to_add)
```

Listing 5.5: Saving all snapshots with a bulk operation

Figure 5.6 illustrates both approaches in comparison. Two bar charts can be seen per commit, the left one shows the original measurement (as already seen in Figure 5.5) and the right bar shows the new measurement which was recorded after all database write access was implemented by using only a few bulk operations. The performance gains are immense, which can be explained by the reduced number of database accesses thanks to the bulk writes. The measurement also supports the hypothesis that the critical factor here is the number of times the database is accessed, and not the amount of data that has to be written - at least for the small amounts

[30] https://pyarango.readthedocs.io/en/latest/collection/#pyArango.collection.Collection.bulkSave, last accessed on 09.02.2020

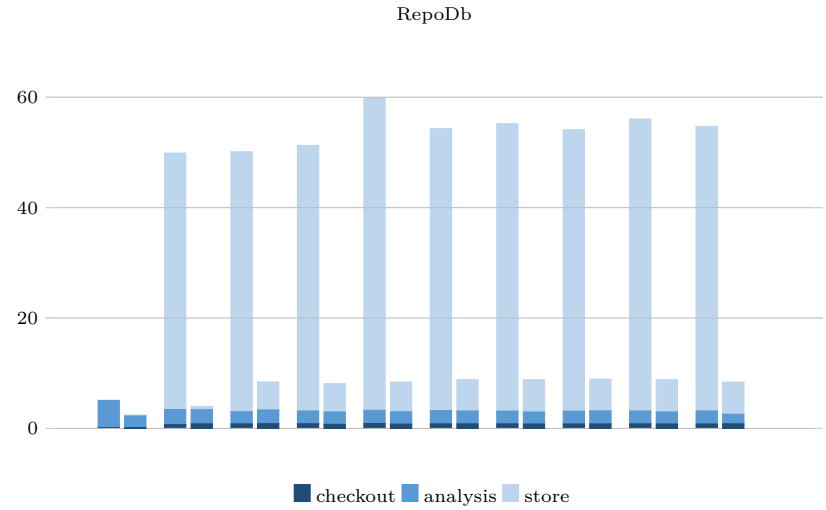


Figure 5.6: Execution time of different mining phases (in sec) for the first ten commits of RepoDB with single (left) and bulk (right) write operations.

that were collected during the first ten snapshots. Measurements of later snapshots with several thousands of files have indeed shown that the total amount of data can also have an increasing impact on the execution time.

To further improve the performance in that case, instead of writing all data to the database at the end of each snapshot analysis, all metric values and entities were cached in memory and were only written to the database once after all analyses were completed. This reduced the database round trips to a minimum, but also required a significant amount of memory. While this had not been a problem during the prototype development and evaluation, it could become problematic in case of very large repositories. To deal with this possible risk, further research would be necessary to check whether a suitable balance between caching and database access can be found.

5.4.5 Workarounds

Most libraries and technologies used for developing the prototype were stable, actively maintained and also had a quite sophisticated and detailed documentation. Nevertheless, it happened that API limitations or possible library bugs did require some workarounds to achieve the expected results. Some of these necessary measures will be briefly documented in the following section.

One workaround was required due to a limitation of the PyArango package: The library offers some functions to greatly simplify the interaction with an Arango database, especially the Graph.link(...)[31] method allows for an easy creation of an edge document between two node documents. However, this method has the disadvantage of not allowing to set the edge's key explicitly, instead when calling it, the database generates and assigns a unique key to the edge document automatically.

[31] `https://bioinfo.iric.ca/~daoudat/pyArango/graph.html?highlight=link#pyArango.graph.Graph.link`, last accessed on 18.02.2020

While, for most cases, this might be the desired behavior, there can be exceptions where assigning a document's key manually has some advantages. For example, manual keys makes it easier, to check whether a corresponding edge with the specific key does already exist in the database. Otherwise, one would have to look at the start and end node of an edge to uniquely identify it, which would generally be possible in non-multi graphs, but requires a bit more effort.

A possible workaround for this limitation was found in a discussion on StackOverflow[32] where it was suggested to create the edge document by using the more generic document API and setting the relevant edge attributes explicitly. Listing 5.6 illustrated the required steps: First, all necessary attributes are stored in a Python dictionary - keys with leading underscores describe ArangoDB specific edge properties - and by using this dictionary, a new edge document is finally created and saved.

```
# set the attributes of the new edge in a dictionary
edge_attributes = {
        "_key": edge_key,
    "_from": portfolio_vertex._id,
    "_to": metric_vertex._id,
    "threshold": threshold_value
}
# create the edge by using the attribute dictionary
edge = self.database[edge_collection_name].createDocument(edge_attributes)
# save the newly created edge to the database
edge.save()
```

Listing 5.6: Creating an edge document manually

It should be mentioned that this way of creating edges was only used for the few master data (for instance, relations between repositories, portfolios or metrics). Relations between metric values and entities, which were generated during the analysis phase were written by bulk operations due to performance reasons (see section 5.4.4).

Another workaround dealt with the execution of Understand's command line tool *und*. Most of the time the tool worked as expected, but there were cases when the application crashed during a snapshot analysis for no apparent reason. These crashes even occurred non-deterministically at random snapshots and could therefore not be reproduced reliably. This behavior might indicate a program-internal race condition [59] where several threads compete for accessing a specific resource, and fixing this issue without access to the underlying source code was not possible. However, this error occurred only rarely and in general went away when the same snapshot was analyzed several times. The algorithm given in Figure 5.1 illustrates the fault tolerance procedure

[32] https://stackoverflow.com/questions/55761897/pyarango-create-edge-with-specified-key, last accessed on 18.02.2020,

that was used for executing the external *Understand* subprocess.

Algorithm 5.1: Handling subprocess errors when analyzing snapshots

```
limit number of tries to avoid endless loops;
max_tries ← 3;
for i ← 0 to max_tries do
    try:
        call Understand sub process to analyze snapshot;
        result ← subprocess.checkout_out(...);
        here, analysis was successful, so process results;
        return success;
    except subprocess.CalledProcessError:
        error while calling sub process, continue with loop;
        pass;
    end
end
if loop ends regularly, all tries failed;
return error;
```

By using a loop, the sub process was called until it either delivered a result or the maximum number of attempts was reached. If the external process exited successfully, the loop was aborted and the program continued as expected. In case the sub process returned a non-negative result, Python's `subprocess.check_out`[33] function threw an exception which was handled accordingly. Since the process could in theory crash infinitely often - which would lead to an endless loop - a loop counter was used to limit the number of retries. If the maximum number of retries was reached, this specific snapshot was ignored and processing continued with the next available snapshot instead.

5.5 Evaluation

The functionality and performance of the system was validated through a single-case mechanism experiment [24]. Here, a predefined set of input parameters were fed into the system and the execution behavior based on these parameters was recorded, as was the resulting output. Both, the measured execution time and the output data were then validated against previously defined expectations.

5.5.1 Input Parameters

Specifying the input parameters was done through the system's configuration options (see section 5.4.4), and here, especially two parameters are of particular interest:

[33] `https://docs.python.org/2/library/subprocess.html`, last accessed on 18.02.2020

The snapshot sample size defines how many snapshots of the target repositories should be analyzed. Since most repositories consist of several thousand snapshots and analyzing each of them would require too much time and resources to be of practical use, this parameter can be used to limit the number of snapshots that will be analyzed to a reasonable amount. But while choosing a high value would result in high resource consumption, taking a value which is too small could result in an inaccurate reflection of the real trend, as some important trend developments might be missing. For this experiment, based on the available hardware (see below), a sample size of *50* was used, which means that 50 subsequent snapshots were chosen from the repository in that way, that the samples' indexes were evenly distributed among all available snapshots. While this number might be sufficient for evaluating the functionality of the mining tool chain, during real world production scenarios, a higher number would be better suited, provided that corresponding hardware is available.

The composition of the software portfolio to analyze on the other side is the second relevant input parameter. Due to the lack of a real software portfolio (according to the definition from chapter 2), an artificial portfolio composed of a selection of Apache Software Foundation[34] (ASF) projects was chosen instead. These projects have in common that most of them are subject to similar quality standards and guidelines and are therefore not that far away from the actual portfolio idea, even if some conditions (such as the competition for shared and limited resources) are not always met. The following criteria were used to narrow down the set of chose projects:

- **Project Size** To estimate the code size of a GitHub repository, the Chrome Plugin *Github Gloc*[35] was used. A pretest with a larger repository of approx. 700,000 lines of code[36] revealed that the execution time of the analysis could take up to two hours and more. While for production use this might very well be considered as a reasonable amount of time, in the scope of this research, smaller projects were chosen to reduce the overall processing time of the experiment. Therefore, four projects with a code size between 100,000 and 200,000 were finally chosen for the experiment.
- **Language** Since the current metric analysis focuses on Java and C# languages, the chosen projects have to be mainly implemented in these languages. While finding adequate Java projects was quite easy, at the time of this writing, there are not many C# projects supervised by the ASF. One of them, *Lucene.net*[37] would have been an interesting candidate, but its sheer size of 2 million lines of code is beyond the scope of this investigation, so the much smaller *log4net* project was chosen instead. This also had the advantage that it metrics could directly be compared with its Java counterpart, *log4j*.
- **Popularity and Topicality** To ensure that the projects reflect the current state of the art and are not just experimental prototypes, two metrics were considered: (1) The total amount of stars the project received on GitHub, which can be used as an indicator for the popularity of the project and (2) the project should ideally be under active development (the last commit should not be older than a few weeks). Unfortunately, the second metric could not be applied in all cases, especially since the development of *log4net* was abandoned in April 2020[38]. The situation was similar with log4j which has been replaced by its successor

[34] `https://www.apache.org/`, last accessed on 14.05.2020
[35] `https://chrome.google.com/webstore/detail/github-gloc/kaodcnpebhdbpaeeemkiobcokcnegdki`, last accessed on 14.05.2020
[36] Apache Tomcat - `https://tomcat.apache.org/`, last accessed on 14.05.2020
[37] `https://lucenenet.apache.org/`, last accessed on 14.05.2020
[38] `https://logging.apache.org/log4net/`, last accessed on 14.05.2020

log4j2[39]. Unfortunately, the second version is of much larger size and was therefore not suited very well for the experiment, therefore the previous version *log4j* was used instead.

The final composition of the portfolio is described in table 5.2.

Repository	Description	Lines of Code	Stars	Analyzed commits
log4j[40]	Java based logging framework	127 K	704	1,5 %
log4net[41]	C# port of log4j	129,4 K	459	4,7 %
Maven[42]	project management and build tool	191 K	2,1 K	0,5 %
RocketMQ[43]	distributed messaging platform	173 K	10,9 K	3,75 %

Table 5.2: Selected ASF projects used for evaluation

5.5.2 Execution

The experiment was carried out on a MacBook Pro 13-inch, Mid2012 with a 2.5GHz dual-core Intel Core i5 processor and 16GB DDR3 Ram with macOS Catalina 10.15.4 installed. Although the basic system configuration might look a bit outdated, the system performs better than one might expect, thanks to several hardware upgrades. Nevertheless, for a real-world production scenario, much more powerful hardware would certainly be used.

After the aforementioned input parameter configuration, the system was started by using a Python 3.7.3 interpreter[44]. The execution time of the analysis of all snapshots and writing the results to the database took almost 45 minutes, plus an additional time of approx. 10-15 minutes for additional post processing operations (calculating aggregations and distributions), which was not measured separately. During the analysis, the ArangoDB server process occupied approximately 6 GB of main memory, which might also include additional swap space.

As a result of the analysis, the following number of documents were written to the database (among others):

1. A list of aggregated metrics per snapshot: 1200 documents
   (50 snapshots * 4 repositories * 6 metrics to analyze)
2. A vertex collection containing all analyzed artifacts per repository: 42000 documents
3. An edge collection storing the metric values for every analyzed artifact per snapshot: approx. 1,6 million documents

[39] `https://logging.apache.org/log4j/2.x/`, last accessed on 14.05.2020
[40] `https://github.com/apache/log4j`, last accessed on 13.05.2020
[41] `https://github.com/apache/logging-log4net`, last accessed on 13.05.2020
[42] `https://github.com/apache/maven`, last accessed on 13.05.2020
[43] `https://github.com/apache/rocketmq`, last accessed on 13.05.2020
[44] `https://www.python.org/downloads/release/python-373/`, last accessed on 14.05.2020

After the analysis, some exemplary AQL queries were executed against the database to evaluate the system's reaction time when accessed by the visualization prototype later. These queries were:

- *Selecting the aggregated lines of code for a given project over all snapshots, grouped by metric.* This query was also used frequently later as it provides the underlying data for visualizing the trend graph of a project for a given metric. As expected, this query executed quite fast because of the precalculated accumulated metric values. The execution time was below 50 ms.
- *Requesting all file artifacts together with their lines of code metric for all repositories for a given snapshot, sorted by metric value and grouped by snapshot and project.* As a result, this query provides a list of files and their corresponding lines of code for every repository. Since this query requires several join operations between the snapshot, artifact and metric-value collection, the execution time was considerably higher and was about 400 ms, which is still in an acceptable time range.

Verifying the correctness of the metric values is difficult because for that, the values calculated by *Understand* would have to be compared with those of another tool, but as studies have shown, these calculated metric values can differ among various tools [14]. Therefore it had to be assumed at this point that *Understand*'s calculations were correct. However, the consistency of the data could be evaluated by randomly comparing entries from the log files created by *Understand* with the corresponding data records. In this way it could be verified that the links between the metric values, their artifacts and the snapshot at which these values were recorded, were set correctly.

5.5.3 Summary

With the help of this experimental arrangement, it was possible to evaluate the functionality of the mining toolchain against the previously specified requirements under mostly realistic conditions. As required by *M.R3*, the process was able to run without any user interaction after the input parameters were specified, allowing for long running batch processes. Regarding *M.R5*, possible parsing errors during the analysis were ignored and the process terminated successfully after all repositories were analyzed. While the execution speed on the given hardware was sufficient for experimental use, analysis of larger repositories (500,000 lines of code and more) or a higher snapshot sample count could require more powerful hardware. Nevertheless, the thresholds mentioned in requirement *M.R4* could be met. Likewise, the amount of data generated did not pose unexpected challenges to the ArangoDB database server, and the execution of selected sample queries have also shown that the system offered a reasonable response times of less than a second.

CHAPTER 6

Visualization

The following chapter gives an overview of the prototypical expert visualization system. To further specify the system's requirements, additional expert interviews were executed to complete the fundamentals described in Chapter 2. Based on these requirements, the actual development process is then explained step by step, starting with a technological review of the used technologies, up to concrete architectural and implementation decisions. The chapter concludes with a brief introduction of the features that were ultimately realized as part of this prototype.

6.1 Questionnaire

Additional requirements and needs were elicited by qualitative expert interviews. For this, a questionnaire was first designed and three experts in the area of software quality management were subsequently interviewed based on this questionnaire. Their responses were analyzed and evaluated by applying systematic thematic analysis [28]. The following section will give an overview of the survey design and interview process and concludes with a summary of the findings from these consultations, a full copy of the questionnaire can be found in Appendix 15.

Study Design

The questionnaire design was based on the recommendations from [26], and the questions were grouped in four main categories. Every category contained both open and close questions and covered a specific aspect regarding information needs or functional requirements which would be relevant for the expert prototype. The four main categories of the survey were:

1. **General questions**
   This block was about gathering some general knowledge to better identify the different roles of the participants and also to better define the possible function scope of the prototype. Questions in this block were, for instance, *„What is your current job description?“*, *„Which software quality related tasks do you have to accomplish during your daily work?“* or *„How many repositories are on average part of the portfolio you manage?“*

2. **Software Portfolio Quality Tools**
   Besides checking which of the tools introduced in Chapter 3 were known by the interviewees, questions in this section were also aiming at identifying tools that are used during the software quality process, together with their advantages and drawbacks.

3. **Visualization**
   In this block, different hypothetical software quality analysis features were presented to the participants and they had to evaluate the usefulness of these features. The presented functionality was organized hierarchically, starting at metric comparison and analysis within a single repository and moving then up to the portfolio level. Some of the questions in this section were: *„How important is it for you to compare the trends of specific metrics on a portfolio level?“* or *„How important would you rate a health indicator that signals the current quality state of a portfolio?“*.

4. **Metrics**
   The last section contained questions that were asking about common metrics used by the interviewees and whether they would be interested in different accumulated or aggregates metric values.

During the pilot phase, a first draft of the questionnaire was presented to an domain expert to validate the overall quality of the survey. This evaluation revealed that especially questions concerning the hypothetical features could benefit from additional illustrations to better explain the mentioned concepts. Also, textual definitions for some commonly used terms were added to the questionnaire as well, since some concepts used in the survey were not always ubiquitous. Care was also taken to formulate the questions as neutral as possible to avoid influencing participants in any way [26].

The final survey was carried out with three experts in person and each questioning took approximately one hour. Every question of the survey was read out to the participant and was also presented in digital form, so that any possible uncertainties could be discussed accordingly. During the interview, the answers of the participants were written down simultaneously on a printed version of the survey. Any comments made by the interviewee were also discussed and noted fur further analysis.

Evaluation

Among the interviewees were a software architect, a software engineer/quality manager and a software developer with additional software quality related teaching activities. All participants worked primarily as software developers or architects, but according to their own statements, had also many tasks related to software quality management and analysis.

For the final evaluation, all transcripts were first read actively in detail, and then semantic codes were developed and assigned to the corresponding answers or text blocks. The determined codes were then continually refined and, if necessary, further adjusted based on additional statements. This code creation and tagging process was done with the help of the commercial data analysis tool MAXQDA[1]. A total of approx. 13 codes could be derived from the answers, and based on these codes, three overarching topics could be identified. All topics were chosen in that way that they were directly related to the research question *„What information needs do software quality managers have when analyzing the quality of software portfolios?“*. Figure 6.1 presents a hierarchical structure that describes the relation between the research question, its themes and their corresponding codes.

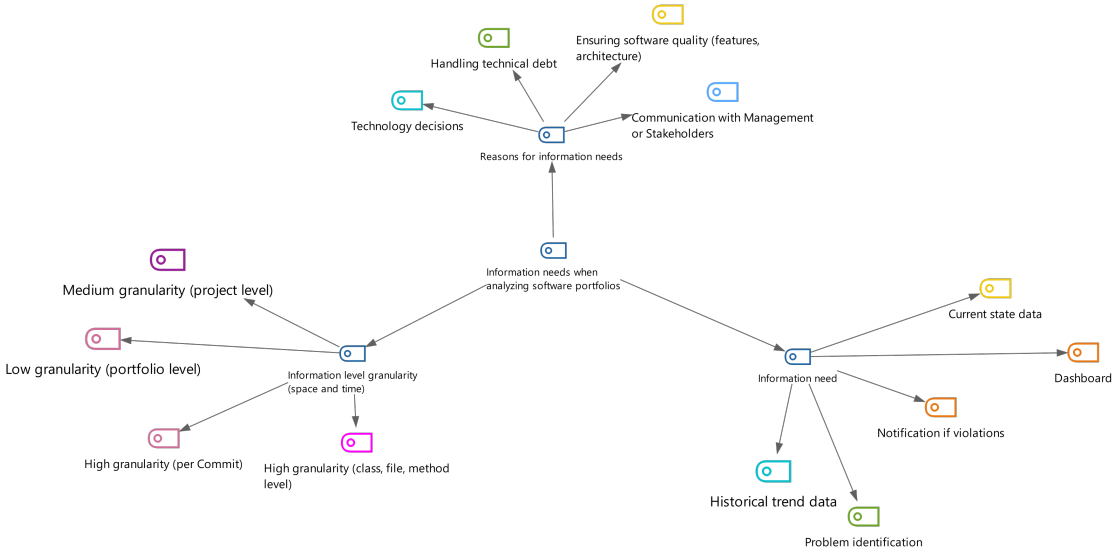


Figure 6.1: A hierarchical illustration of the themes and codes that were identified during the thematic analysis of the survey.

[1] https://www.maxqda.com/, last accessed on 21.03.2020

The identified themes were:

- **Possible reasons for information needs**
  Several reasons for information needs could be identified. Participants mentioned that they require information to evaluate the current quality state of their software repositories, for instance, to fight technical debt, but also to measure the effectiveness of possible quality measures.

  Two participants also believed that analyzing quality metrics could play a critical role when evaluating new technologies (like frontend libraries or frameworks). Since all of these technology candidates have a similar application profile and scope, they could very well be considered as parts of a single technology portfolio. Comparisons within this portfolio could therefore provide meaningful information.

  Furthermore, communication with other stakeholders, be they decision-makers or customers, is an another important reason for obtaining information. The gathered data could help in communicating the project status to all participants, but could also serve as a quality indicator to meet specifications and guidelines from external clients. But, according to the respondents, this application is not without controversy, as single metrics are seldom expressive enough to describe the complex quality state of a software system.

- **Spatial and temporal information granularity**
  Regarding the information granularity, roughly two dimensions can be distinguished. On the one hand, the temporal granularity defines the intervals at which the information should be sampled and made available. Interestingly, all participants agreed on a granularity at commit or snapshot level, as having information available at this sample rate would provide the best support when tracking down possible quality issues. According to the participants, here lies also the problem with the current status quo: In most setups, the quality analysis as part of the continuous integration process takes place only during the nightly build. Depending on the development team's activity, however, numerous commits could easily occur between two nightly builds, which makes it difficult to clearly localize the relevant code that was responsible for the change of a quality metric. Unfortunately, calculating quality metrics is a time consuming process that could increase the build time significantly. Therefore, analyzing every check-in in real-time as part of the CI process was not considered a practical solution by the participants.

  But beside this temporal information density, also the spatial information granularity, i.e. the hierarchical levels at which quality metrics are recorded, plays also an important role. In general, three different layers could be roughly distinguished here during the thematic analysis, starting at individual code artifacts, over project-wide metric values up to metrics gathered on portfolio level. If one looks at the information needs of the interviewees here, a much more differentiated picture emerges. Because comparisons on project level within the same portfolio is often not supported by most of today's commercial tools, two participants would

see the greatest additional benefit if such a functionality would be provided. In contrast, the third interviewee would see only little benefit here. However, he agreed that, while a mere comparison of metrics at portfolio level would not be very meaningful, the reason for any possible metric fluctuations between projects might indeed provide interesting insights.

The participants agreed that comparing different metric trends at individual project level would be a very useful feature, since the correlation of different metrics could lead to interesting insights that would be helpful for identifying potential quality problems. On the other side, comparing metrics of different levels of granularity, like comparing inheritance depth of a single class with the average inheritance depth of all classes of a project, was considered to be of minor importance. One participant argued here, that metrics at class or method level are too fine-grained to be meaningfully analyzed in the overall project- or portfolio context. In his opinion, it would be more interesting to compare the metrics of different modules which each other, to see how they diverge over time. One expert noted that it is relatively difficult to make general assumptions here, since the added value which could be gathered through these comparisons would highly depend on the metrics chosen. If suitable metrics are chosen, it could even make sense to compare metrics on class or method level, especially if additional metric thresholds are used during the analysis.

Another point where all participants would see a great benefit would be the use of a health indicator that provides a quick and clear overview of a project's or portfolio's overall quality condition. However, to provide meaningful information, the criteria of this indicator should be freely configurable and ideally be individually customizable per project, as, for instance, system-critical projects would require different limits than less critical components, but both would contribute to overall portfolio stability, albeit to different degrees. Instead of using a health indicator, one interviewee suggested to better speak of a *problem indicator*, which purpose would be to notify the user as soon as any relevant quality characteristic reaches a critical point.

- **Concrete information needs**
  Since the specific information needs of software quality managers are an essential part of the research question, the answers the interviewees gave in this area form the third theme that was identified during the thematic analysis. The participants expressed both interest in the current quality status of the portfolio, but also in historical metric trends. All three experts mentioned that they would monitor these two sources of information in order to be informed in case any possible quality violations would occur. In the case of individual quality snapshots, a possible reason for such a notification could be if, for instance, a calculated metric value falls below or above a certain threshold. In contrast, when looking at entire trends, it would be important to find out whether the curve is approaching a defined boundary value, even if it has not yet been reached. But regarding this point, the participants again

emphasized, that these limits can be very project-specific and therefore should be individually configurable.

Another important demand concerns the quick identification and evaluation of possible problem areas within the code base. It would be very useful here, if the system could detect and highlight possible hotspots and would also allow an easy navigation to these points. These hotspots, in turn, could be of various types, from trend changes due to merge operations to longer build times because of performance problems during test execution, many different scenarios would be conceivable here.

When asked about possible metrics and their characteristics, most participants preferred the standard metrics provided by SonarQube, such as test coverage, code smells, number of bugs by severity, etc., but occasionally interest was expressed also in object-oriented or complexity metrics. However, the participants agreed that a single metric alone is of limited significance and that the combination of different metrics is of special importance for further analysis. Regarding the aggregation of single values, both minimum/maximum boundaries as well as distributions of these values was rated as most important, while aggregated average or median values were considered to be of less significance.

In addition, two participants emphasized the importance of a dashboard that would present the most relevant information in an easy and compact way. In that way, the most necessary information needs could be served in a very quick and efficient way.

Summary

Even if the number of samples was relatively small with only three experts that were interviewed, some of the results partially support the previously examined observations. The following summary is therefore intended to provide a concluding overview.

Most participants expect that a combination of different metrics will be more meaningful than individual metrics. This finding is consistent with previous studies [15], although it was emphasized that the actual benefits may depend heavily on the metrics chosen.

The participants identified test coverage, technical debt (in days) and the number of code smells and bugs as important metrics, but these statements should be considered with a bit caution: All three interviewees were mainly using SonarQube for their analysis and the mentioned metrics are all part of SonarQube's standard metric palette, so their answers might be a bit biased here. Nevertheless, there exist some overlappings with the metrics identified by Buse and Zimmermann, although in their survey, they used the more wider term *indicators* instead of metrics [15].

Accumulated average values for metrics were only considered to be of limited help, instead the participants expect more benefits from min/max values or metric distributions. While the need for aggregated data is also mentioned by other sources [16], no explicit definition of how these data should be aggregated could be found there.

The same applies to the ideal sampling rate at which the analysis should take place, an aspect which was not examined, or at least not mentioned, in other papers. Here all interviewees favored a rate based on commits- or snapshots in comparison to the current daily (or better *nightly*) build rate.

The availability of a health or problem indicator together with some kind of alert system was both mentioned during the survey and could also be found in the research literature [4]. The same applies to the ability to identify and navigate to problematic parts in the code [15].

Finally, while analysis on different artifact levels could also be identified in literature [15], a project-wide analysis on portfolio level was only mentioned by the survey's participants. But this might be related to the fact, that most of the examined papers were focusing their scope only on project- and not on portfolio level.

Threats To Validity

Before concluding this section, it should be pointed out again that the number of participants was relatively low and therefore cannot be regarded as representative of the target group. Likewise, there was a strong preference in the sample with regard to SonarQube as the quality tool of choice, which might have, for instance, affected the answers regarding the preferred metrics.

Likewise, although all interviewees performed quality-related tasks in their roles, none of them explicitly identified themselves as decision-makers in the area of quality management, which might differ somewhat from the roles described in the researched literature [2, 15]. However, as already mentioned at the beginning of this section, the stakeholders in this area are a rather inhomogeneous group anyway.

It should also be regarded that the expert who helped refining the questionnaire during the pilot phase, also participated in the actual survey, which might make the questions more tailored to her personal experience than it would be the case with the other participants.

To prevent possible influencing here, the questions were formulated in such a way that answers could be given as openly as possible [61] and a lot of scope was also allowed for further discussions during the interviews.

6.2 Requirement Specification

Based on the previous research, two different sources could be used to obtain requirements for the expert visualization prototype:

1. A majority of the requirements could be derived from the information needs that were either identified through literature research (Chapter 2) or by evaluating the qualitative expert surveys (Section 6.1). Implementing these requirements would therefore also provide the user with the demanded information.

2. There also exist dedicated studies regarding requirements of software visualization tools. While, for instance, the research in [62] is not specifically related to software portfolio analysis, but instead investigates the more broader domain of software visualization and analysis tools as a whole, some of its findings should also be taken into account here.

The following requirement list is not exhaustive and there certainly exist many more, both functional and non-functional requirements, but due to time constraints, not all of them could be implemented in the prototype, so a pre-selection had to be taken. While this selection tried to focus on the requirements that were identified as most relevant by either the literature, or during the expert interviews, a certain degree of subjectivity will always remain for such decisions.

V.R1: Easy Usage
Most stake holders are no data science experts and also might not have very deep knowledge in this area. Therefore, the visualization system should have a clean user interface and should be easy to use, even for non-domain experts [15]. As it can be assumed that the users will have experience in using common application software like project management tools, development environments, word processors and others, the visualization prototype should follow common guidelines for user interface design and use visualization and interaction concepts that are familiar to the end users. Furthermore, additional concepts like tooltips or descriptive labels could also improve the usability of the prototype, but more complex concepts like on-screen tutorials will not be realized because of time and resource constraints.

V.R2: Allow for Fast Analysis
Software quality managers often can spend only limited capacities on their analysis tasks, therefore the analysis tool should regard their time constraints as much as possible [16]. To achieve this, the system should provide highly aggregated data and aggregating this data should not take too long (while the concrete time might depend on the amount of available data and the final query that is executed by the user). Summarizing the metric and trend data is a functional requirement that could mostly be met already during the data mining process. However, providing fast response times during the data visualization might affect several parts of the system architecture and should therefore also be considered as a non-functional requirement. Regarding the type of accumulated data, the interview analysis revealed that min/max values and distributions, together with possible threshold violations might provide the most benefit for the users. Aggregated median values were also higher rated than averages or other aggregated values.

V.R3: Combining Different Artifacts and Metrics Data
Although providing too much information might be problematic, insights could only become immanent if various artifacts and metrics are linked with each other. For this, it is necessary to have a wide palette of possibilities to choose from [15].

Users should therefore be able to choose the artifacts they want to analyze and also select the metric values they want to monitor. To allow for better comparing the metrics of different artifacts with each other, it should be possible to group the various metric trends together in a single diagram.

V.R4: Reduce Information Overload

Especially larger portfolios that are monitored over a longer period of time can produce huge amount of data. Presenting this data in an unfiltered way to the user can soon result in information overload (also referenced as *overplotting* [62]). This makes it very difficult to gather meaningful insights from the data.

However, reducing data without loosing too much information is not a trivial task and there is a high risk that important information might be accidentally removed from the result set. To support the user in choosing the information which is relevant for him, the system should provide different functionality like filtering or zooming that allows the user to focus on the important data [62]. Some of these concepts will be introduced in section 2.4.

V.R5: Focus on Historical and Current Data

As stated by Buse and Zimmermann [15] and also by the participants of the survey, there is a clear preference towards historical data and trend analysis. Visualization should therefore focus on these aspects. Section 2.4 presents some visualization concepts and analyzes their capability of illustrating historical trend data. Unfortunately, adding large trend data can increase the complexity level of a visualization significantly, therefore ways to reduce the possible information overload must also be examined in this context.

V.R6: Different Levels of Details

As already mentioned, quality managers are interested in both, high-level project overviews, but also in investigating single project artifacts [15]. The expert interviews seem to support this finding, especially regarding analysis on portfolio level. To achieve this, the visualization prototype should both provide different views depending on the chosen detail level and should also allow the user to easily navigate between these different views [62]. Depending on the currently selected detail level, different visualization concepts might be applied.

V.R7: Notification System

Getting notified if an unexpected event happens is also an important information need, the system should therefore provide some kind of alert notification if specific conditions (like thresholds) are met [4]. According to the interviewees, the notification criteria should best be configurable on a per-project basis. While in a commercial application, this feature might also include support for external notifications, like sending e-mails or Slack[2] messages, in the context of this work, only visual alerts will be taken into consideration.

[2] `www.slack.com`, last accessed on 22.03.2020

V.R8: Health Indicator
A health indicator provides functionality similar to the ones of an alert system: With the help of this indicator, a user gets an immediate overview of the current quality state of a specific artifact [60]. This feature can be problematic, since the current state of an artifact can be complex and might also depend on the context, so a single value is often not sufficient enough to express this information. But nevertheless, such a feature could at least be helpful in indicating a general problem that would need further investigation. Again, the participants of the survey suggested to make this indicator configurable to better reflect the individual information needs for each project. Some possible health indicator concepts will be introduced in section 2.4.

V.R9: Overall Portfolio Overview
To get a quick overview of the portfolio state, a possible visualization system must allow users to view and analyze the overall portfolio and its parts in an easy way. This feature could best be combined with the aforementioned *Level of Detail* feature, so that users could, for instance, start at the overall portfolio view and from there drill down into selected projects and entities. In this context, two of the interviewed experts suggested some kind of dashboard, where the most relevant metrics are immediately visible at a glance. Since designing a comprehensive dashboard can be considered a science on its own right and is not a trivial task [63], in the scope of this work only a minimal dashboard solution will be developed, but this could definitely be an interesting point for further research.

V.R10: Stakeholder Specific Information Support
Since the group of stakeholders is very inhomogeneous, so are their information needs. Buse and Zimmermann [15] therefore suggest that possible analysis tools should take this into account and should be tailored to the individual needs of the stakeholder. Also Kienle and Müller come to the conclusion that customizability is a very important feature for a wide range of users [62]. Providing this feature through a plugin-interface would be possible, but also too complex in the scope of this work, but letting the user customize the metrics and thresholds which are relevant for the analysis could be a suitable compromise to implement.

V.R11: Code Proximity
This feature links the current visualization directly to the source code that was analyzed and allows the user to directly jump to the repository and browse through the relevant sources [62]. Many tools (like SonarQube - see Figure 6.2) provide some kind of overlay where the user can see both, the analysis result (like code smell, duplication, test coverage etc...) and the corresponding source code in one single view.

[3] `https://www.sonarqube.org/`, last accessed on 02.03.2020

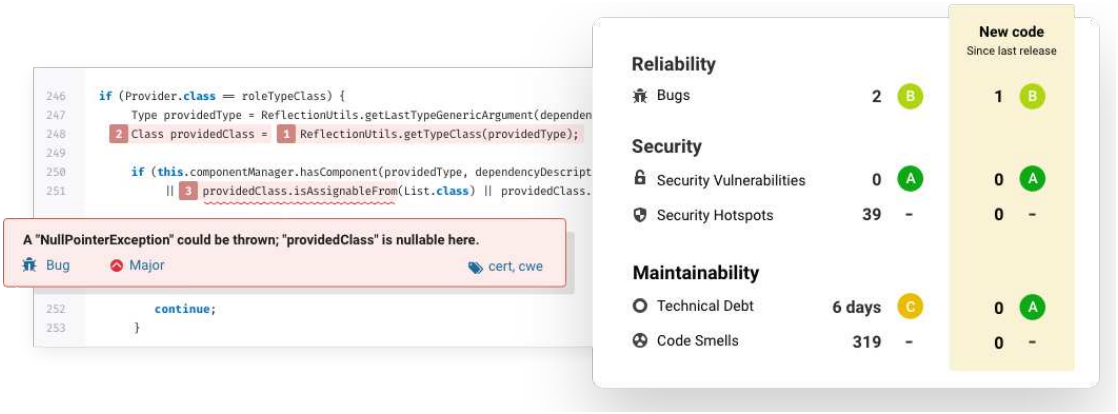


Figure 6.2: SonarQube supports easy navigation between source code and analysis data[3]

Studying this list reveals that there are some conflicting requirements. For instance, the system should on the one hand avoid information overload and provide highly aggregated data, but on the other hand, the user needs access to as much artifacts and metrics as possible to identify any potential relations between them. Also, while the prototype should be easy to use, it should also provide sophisticated functions for navigating between different detail levels and let the user customize its analysis process individually. But these contradicting requirements are a very essential part of this work: One of the main purposes of this visualization prototype is to find out how a possible implementation could balance these requirements against each other and where compromises have to be made to generate the most benefits for the users.

6.3 Implementation Overview

The expert visualization prototype consists of two separate and decoupled systems, which will be called *frontend* and *backend* in the further course of this work. The frontend part is a web application that is responsible for data visualization and retrieving user requests. These requests are sent to the backend application via a HTTP based REST API, which will then process each incoming request, loads the required data from the database and sends the resulting records back to the frontend.

The following sections will give an overview of these two systems, both of their overall architecture and their implementation details. It will start with an overview of the selected technologies and then goes on to describe in deep how specific features were realized.

6.3.1 Technology Stack

Both systems serve different use cases and therefore also have different requirements regarding their implementation. Thus, the technology stack for each system was chosen

and composed individually to best meet these requirements. In order to allow for decoupling between the frontend and backend part, a web-based solution was chosen. This decision resulted in the only restriction that both parts have to agree on a common communication protocol, which, for web applications, is most often the HTTP/HTTPS protocol. Besides this limitation, all other technologies could be freely chosen.

Backend

Starting with the backend and regarding the fact that the application has to communicate with the same database as the mining process (see chapter 5), it was an obvious decision to adopt the same technology stack. Therefore, **Python** was again the programming language of choice, not only because the same database access wrapper (*PyArango*) could be use for both applications, but also because Python has a wide portfolio of sophisticated data-science libraries, like Numpy[4], which would become very useful for later data processing.

For implementing the REST API in the backend, different web-frameworks and libraries were considered: Django[5], for instance, is a very popular Python web framework but seemed a bit oversized for the current work, as only a small part of the framework would be relevant for implementing the REST API. Therefore **Flask**[6], which is a more lightweight web application framework, seemed to be a better choice here, especially since the **FlaskRESTful**[7] extensions provides additional support for implementing minimal REST APIs.

While Flask provides basic web server functionality, in order to implement a fully-fledged web application, additional libraries are needed. One main aspect is the handling of dependencies between the different parts of the application, for which **Flask-Injector**[8], was used. Flask-Injector is a dependency-injection library which allows for decoupling different components from each other by inversion of control. All relevant dependencies are registered in a dependency container, and whenever a dependency is requested by a component, the container is responsible for resolving it. In that way, coupling between different implementations can be reduced, which can improve the maintainability of the application significantly [72].

Another important task within web applications is parsing and validating incoming web requests. For this, **webargs**[9] was used, which is a very easy to use HTTP request object parser. Listing 6.1 demonstrates the usage of webargs for parsing incoming request objects. First, the expected structure of the incoming payload has to be defined (line 1-6), and after that, the corresponding REST method should be decorated accordingly (line 8). In case of a valid incoming request, webargs will then map every incoming parameter

[4] `https://numpy.org/`, last accessed on 05.03.2020
[5] `https://www.djangoproject.com/`, last accessed on 05.03.2020
[6] `https://palletsprojects.com/p/flask/`, last accessed on 05.03.2020
[7] `https://flask-restful.readthedocs.io/en/latest/`, last accessed on 05.03.2020
[8] `https://github.com/alecthomas/flask_injector`, last accessed on 10.03.2020
[9] `https://webargs.readthedocs.io/en/latest/`, last accessed on 10.03.2020

to the corresponding field in the data structure.

```
args = {
    """structure of incoming HTTP request objects"""
    'metricId': fields.Str(),
    'portfolio': fields.Str(),
    'threshold': fields.Decimal(allow_none=True)
}

@use_kwargs(args)
def put(self, metricId, portfolio, threshold):
     """handles incoming HTTP request.
        parameters will be extracted and assigned by webargs parser.
     """
     # implementation omitted
```

Listing 6.1: Parsing request parameters

Frontend

For the frontend part, a choice between different JavaScript based web frameworks had to be made. At the time of this writing, the most popular ones are *Angular*[10] and *React*[11]. Angular is generally considered as more opinionated, as it already contains many relevant functionalities, like dependency injection libraries, routing etc., where React leaves many of these choices to the developer.

By default, Angular uses **TypeScript** as implementation language, a statically typed super set of JavaScript that transpiles to native JavaScript code [73]. Compared to other common OOP languages like Java or C#, it uses structural identity: If an object implements the same methods that are required by an interface, the object can then be considered as an implementation of this interface, even if this fact is not explicitly expressed by the type system. This mechanism makes integration of TypeScript into the JavaScript ecosystem easier, as there, objects are often created on the fly without a formal type definition [73]. Because of existing knowledge due to personal experience, **Angular** was chosen for the frontend implementation, but this decision was mainly based on subjective opinion.

But since a main part of the frontend's functionality is related to drawing charts and diagrams, the availability of possible chart-drawing libraries for the chosen frontend technology was also an important decision criterion. A popular main-stream JavaScript chart library is *D3.js*[12]. Strictly speaking, D3.js is not only for drawing charts and graphs, but instead is a more general data visualization library that allows reading and transforming data from and into various formats. By using this library, developers can read various forms of data documents and transform them into many other forms, among which are, of course, also charts. Regarding the chart types introduced in section 2.4, D3.js

[10] https://angular.io/, last accessed on 05.03.2020
[11] https://reactjs.org/, last accessed on 05.03.2020
[12] https://d3js.org/, last accessed on 08.03.2020

supports all common types like line charts, radar charts and histograms, and even more exotic types like parallel coordinate charts[13]. However, its native JavaScript nature makes integration into an Angular project, while still possible, a bit more extensive, because some additional glue code would be required to integrate the library into Angular's more declarative approach.

Therefore, the choice fell on **ngx-charts**[14], a data visualization library with native Angular support, based on SVG geometries. Ngx-Charts also supports most of the commonly used chart types like line charts or radar charts, although compared to D3.js, its list of available graphs is not as exhaustive. Nevertheless, a missing diagram type could still be added by implementing a custom chart type, as suggested by the ngx-charts' documentation[15], and if this approach turns out to not be feasible enough, D3.js could still be added to the project and both visualization libraries could be used in parallel.

Even when separated from the backend, front logic can become increasingly complex with ongoing project lifetime, especially when several components have to interact with each other and have to access or manipulate relevant state information. Additional state management libraries can help by mastering these challenges. While many popular state management libraries like Flux[16] or Redux[17] have their origins in the React ecosystem, there exist also solutions for Angular. Especially the Redux pattern has an implementation for Angular, called NgRx[18]. While this library might be suitable for most business applications, it might be a bit oversized for this prototype, so a more light-weight state management system, **Akita**[19], was used instead. Section 6.3.3 will discuss this topic in more detail.

The last aspect of the frontend that had to be considered was the CSS framwork that should be used for layouting the application. Again, several solutions were available, the most commonly used ones are:

- **Angular Material**[20] is a UI component library that is especially tailored for the usage within Angular applications. It provides different components for rendering user interface elements, but also some layout components for arranging the content of a web page.
- **Bootstrap**[21], while still a very extensive CSS layout framework, it might become a bit outdated, as parts of it - like extensions or animations - still rely on jQuery.

[13] https://www.d3-graph-gallery.com/, last accessed on 08.02.2020
[14] https://github.com/swimlane/ngx-charts, last accessed on 08.03.2020
[15] https://github.com/swimlane/ngx-charts/blob/master/docs/custom-charts.md, last accessed on 08.03.2020
[16] https://facebook.github.io/flux/, last accessed on 09.03.2020
[17] https://redux.js.org/, last accessed on 09.03.2020
[18] https://ngrx.io/, last accessed on 09.03.2020
[19] https://netbasal.gitbook.io/akita/, last accessed on 09.03.2020
[20] https://material.angular.io/, last accessed on 09.03.2020
[21] https://getbootstrap.com/, last accessed on 09.03.2020

- **Bulma**[22] is a more lightweight alternative to Bootstrap. It also provides different layout options and user interface components, but at the same time does not have as many dependencies as Bootstrap.

Since most CSS framworks are equally powerful and the visualization prototype does not have any extraordinary requirements regarding its layout, the only relevant criteria would be a good documentation and easy integration, therefore **Bulma** was chosen, but the other candidates would have been certainly also good choices.

6.3.2 System Architecture

The visualization prototype uses a classical layered architecture [74], where every layer represents a different concern of the application. In contrast to web applications with static views, the prototype uses a single page application (*SPA*) architecture, where the backend is only responsible for providing the relevant data and the rendering logic is mostly encapsulated in the frontend. This decoupling is also illustrated in the architecture diagram (Figure 6.3), where every part provides its own layers. In this type of architecture, the different layers depend on each other, where in general, a higher layer should only have access to its adjacent lower layer, but never the other way round. This reduces the effort when replacing a specific layer, as dependencies have to be followed only in one direction. While in most cases, replacing an existing layer's implementation with a different one might not happen very often (for instance, one would rarely switch the database from a relational one to a NoSQL database system), but replacing layers with a corresponding test stub for easier unit testing [34] might be a very common scenario. The different layers and their connections, as shown in Figure 6.3, will be briefly described here, beginning with the lowest backend layer and moving up to the final, presentation layer in the frontend.

- **Database** The visualization backend uses the same ArangoDB database access as the mining tool chain, but its focus lies clearly on executing read-only queries, either for plain or accumulated data.
- **Repository** A repository functions as a intermediate layer between the database access and the actual domain logic [74]. In that way, the domain logic can be kept clean of any database specific implementation. While for larger domain models, it is generally considered to provide a single repository per global domain entity [30], for simplicity, this prototype encapsulates all data access behind a single repository implementation.
- **Business Service** This service layer contains the actual business logic and is therefore one of the (if not the most) essential parts of an application [74]. In case of this prototype, the actual domain logic is relatively manageable and focuses mainly on data transformations, like normalization.

[22] `https://bulma.io/`, last accessed on 09.03.2020

- **REST Controller** The backend sided REST controller provides the communication interface called by the frontend It is build by using FlaskRestful resources, where every resource represents a specific, independent concept of the API that can be accessed through common REST methods like *Get* or *Post*. The mapping of a specific resource to an endpoint is finally done by a process called *routing* [75].

- **REST API Service** The API layer placed on the frontend is the other side of the communication and is responsible for sending requests to the backend and retrieving response data from it. All communication is performed via HTTP, additional data is either send via query parameters or as JSON message payload.

- **Application Service** This layer implements the use cases that are triggered through user interaction, by calling domain logic via the REST API. Section 6.3.3 will focus more closely on the application state management that is implemented in this layer.

- **Presentation** The presentation is built upon various Angular components, which together compose the whole application screen. This visualization prototype uses mainly components with two different, architectural roles: *Container Components* (also referred to as *Smart Components*) and *Presentational Components*. While this distinction originates from a pattern mainly used for ReactJS applications [78], it also has advantages when applied in an Angular application. The idea behind this pattern is to distinguish components depending on their function: Container Components normally do not have any visual representation, but instead orchestrate their child components by providing them input data and receiving their output data (via event emitters). In that way, they are responsible for updating the application' state (by calling application services) and propagating these state changes to their children, the presentational components. These components, on the other side, contain as less logic as possible and instead focus only on rendering the relevant application state they receive via data binding from their parents. While this approach requires some additional boilerplate code, it also has some advantages, as it provides a good separation of concern and keeps the view logic separated from the actual presentation, which improves the readability and maintainability of the code.

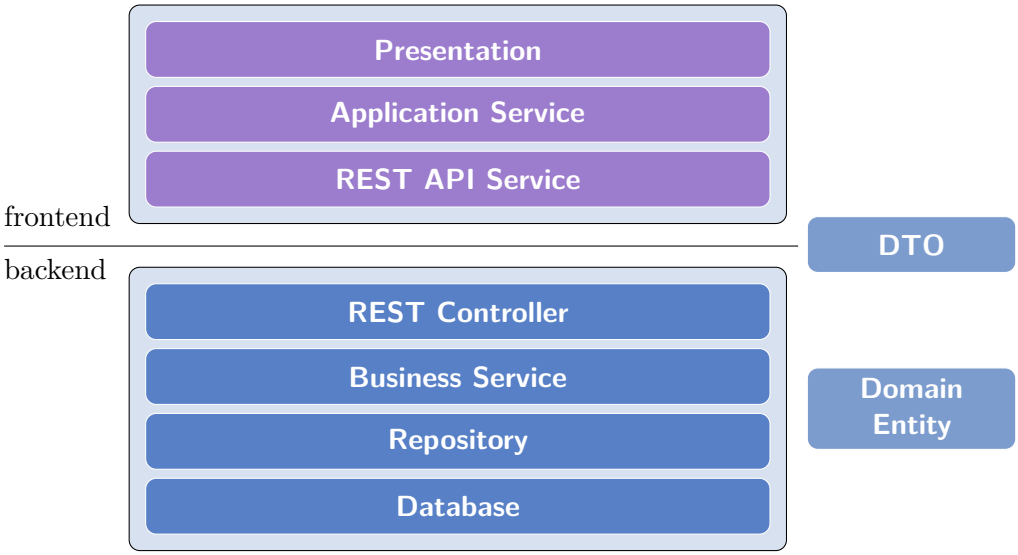


Figure 6.3: Layered architecture of the expert visualization system

In most cases, an implemented feature can be considered as a vertical slice through the whole architecture, as its functionality might affect all layers. For instance, implementing a new diagram chart for visualizing the metrics of of a portfolio might require the following steps:

1. A new Angular component must be added to the presentation layer, this component will render the chart and will also receive any chart related user input. The input handlers of the component will then call service methods in the application layer for further processing.

2. The service in the application layer will call the API layer to request the relevant data. As soon as the data arrives, it will update the application state which will notify the component, resulting in a visual update.

3. After receiving the data request from the application layer, the API layer will transform the input data into DTO objects and will send a HTTP request to the backend. For this, it needs to know the exact endpoint that is responsible for handling the corresponding use case.

4. The API controller on the backend side receives the DTO object and delegates the request to the domain layer. It will also convert the resulting data to a DTO again and send the response back to the frontend.

5. Depending on the requested data, the domain service could either directly call the repository to retrieve the data and return it back to the controller, but it might also be the case that some additional data processing is necessary.

6. The repository will finally access the database to retrieve the data that should be rendered in the frontend view. In most cases, this will be done by executing a predefined AQL script. If necessary, the retrieved document object will be converted into a domain object, if the structure returned by the script is not already sufficient.

While this might sound like much effort for realizing a single feature, the overall complexity of each feature is split up between the different layers which makes each layer-specific implementation smaller and easier to handle. Also, most features can be implemented in isolation from each other, so that adding functionality for a new use case might not require changes in existing features, which reduces complexity further, as potential side effects could be minimized or even avoided. Additionally, testing can also be managed relatively easy by replacing dependencies on lower layers with test stubs.

This section gave an overview of the system architecture that served as base for the further development of the expert visualization system. While many of the presented architectural concepts are mainly used when designing enterprise applications, they can also be applied to reduce overall complexity and improve maintainability for applications of a smaller scale, like the prototype for this expert visualization system. Another important architectural aspect, the handling of the application state, will be discussed in the next section.

6.3.3 State Management

Many frameworks like MVC or especially MVVM make use of a two-way data-binding mechanism, where changes triggered by the view make direct updates to the model and vice versa [79]. While this approach might work for smaller applications, it can become complicated, if, in the case of Angular, different components would update different models, and a single model might also be bound to different views. To simplify the state management in complex UI frontends, the *Redux* pattern suggests the following three steps [79]:

1. There should only be a single state of truth that keeps track of the application state. All components should get their information from this source.
2. The application state should only be read-only. Updating it should therefore result in a new state object reflecting the changes. In that way, all state updates could be tracked back to the initial state.
3. State updates should only be done without side effects. In Redux, these updates are performed by so called *Reducers*.

While these concepts were mainly developed for React based applications, the same approaches can also be applied when developing Angular applications. In case of this prototype, the *Akita*[23] state management library was used to implement a more simplified

[23] `https://netbasal.gitbook.io/akita/`, last accessed on 09.03.2020

version of this Redux pattern. Akita uses many of the aforementioned concepts, but deliberately omits the concept of Reducers in favor over simple application services. A typical application that uses Akita for state management should provide the following three concepts.

Store The store is the single source of truth. It is the only place where the application state is stored. While Redux suggests keeping all state information in a single store, Akita recommends the usage of *Entity Stores*, where each store keeps track of a relevant set of domain entities. Additionally, these stores might also contain UI relevant information related to the entities.

Service A service is the only option to trigger state updates (by calling methods from the store's interface). In that way, components can be kept clean from state changes. This separation of concern could already be found in the system architecture (section 6.3.2), where all application relevant code is located in the *Application Service Layer*. Any possible state updates will also be initiated by this layer.

Query Queries are separate objects that can be used to query store information. All query objects are observable, meaning that a component can subscribe to them to get notified whenever the application state changes. In case of an update, the query will provide a new state object (since the state is immutable) which components can use to update their view. Queries can be configured by using different filter options to return only a specific slice of the whole application state. Thanks to the fact that components always receive a new state instance, their update mechanism can be changed to `ChangeDetectionStrategy.OnPush`, which could result in better performance because then components only have to update whenever their input binding references have changed[24].

Figure 6.4 further illustrates this mechanism: A user interaction will cause a component to call an application service. Depending on the use case, the service will send a request to the backend, and as soon as it receives the response, will update the frontend application state by calling the store's update method. In that way, the backend domain state (which is stored in the database) and the frontend application state will always be in sync. Please note here that the backend domain services should normally be stateless, this allows for easier scaling as several backend nodes could be run in parallel. As soon as the store update is completed, the update will be published by the corresponding query objects. This results in all subscribed components getting notified and re-rendering their template accordingly.

[24] `https://blog.ninja-squad.com/2018/09/27/angular-performances-part-4/`, last accessed on 14.03.2020

[25] `https://datorama.github.io/akita/`, last accessed on 21.08.2020

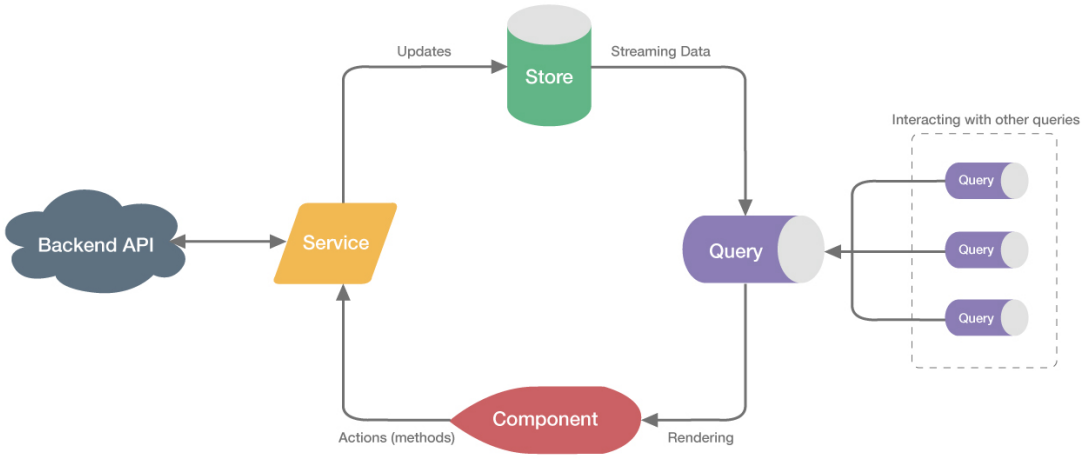


Figure 6.4: Akita State Management Architecture[25]

6.4 Prototype

This section gives an overview of the actual realization of the expert visualization prototype. To support the traceability of all taken design decisions, every feature description will also reference the relevant requirement which it tries to fulfill. The following presentation focuses primarily on the visual components of the prototype, the REST interface which is used to access the underlying backend data storage is mostly self-explained and therefore should only be mentioned briefly if necessary for explaining a visual concept.

Based on the identified needs for a global portfolio overview (*V.R9*), the possibility to also dive into individual aspects of the portfolio (*V.R6*) and the necessity of defining individual health indicators (*V.R8*), the prototype is separated into two different main views, a *portfolio dashboard* and a *snapshot-centric view*, and an additional modal view to define custom health metric rules. Switching between both main views is easily possible through a navigation bar located at the top of the page. These views, together with the modal dialog option for health indicators, are described shortly in the next sections.

6.4.1 Portfolio Dashboard

The dashboard, as seen in Figure 6.5, uses a a tabular presentation for giving an overview of all projects within the portfolio (2). Each column of the tabular view represents a single project, whereas each row provides a configurable selection of all available metrics (3). The intersection of project-column and metric-row results in a historical trend series

of all selected metrics for the corresponding project, visualized in a 2D line chart (4). The displayed time series includes all recorded snapshots for the respective project in ascending chronological order (the right most data point is therefore the most recent one). This feature conforms to the demand of focusing on historical and current data (*V.R5*). In order to keep the information overload of each individual diagram as low as possible (*V.R4*), only one metric is pre-selected per row, but if desired, the user can always select additional metrics and combine their trends in one single line chart (see also requirement for customization *V.R10* in this context). In that way in-project comparisons of metric trends can be done much easier, nevertheless users should be aware of the risk of possible overplotting [62] when using this feature.

Figure 6.5: Portfolio Dashboard View with the following elements: (1) selected portfolio and overall health state, (2) list of projects within portfolio, (3) list of selected metrics, (4) trend visualization of selected metrics per project, (5) sort order of projects, (6) controls for adjusting trend graph visualization, (7) individual project health state

In addition to the selection of metrics, each row also provides an additional control panel (6) which can be used to customize the trend chart (*V.R10*), the following options are there available:

Absolute Values / Relative Changes Due to the different value ranges of various metrics, comparing absolute trends can be very difficult, especially if these ranges diverge widely. To allow for better comparison between metrics, the user can therefore choose between two different options when visualizing the metric values. Option one shows the absolute values as they were recorded during the data mining process. Option two, on the other side, uses the first recorded value as base line and expresses all further values as relative changes against this reference value. This makes comparing metric trends easier, as long as both metrics have at least the same growth rate compared to their base value (*V.R3*). Switching between the two options changes the diagram instantly, as the absolute values are already precalculated and can be retrieved directly from the database. Calculating the

changes against a base value could theoretically also be done in advance, but is currently done on demand. This was motivated by the idea that in a later version, users could have the possibility to choose a variable base index - however, at the moment the base index is hard coded to zero. Regardless of that, the operation itself can completely be swapped out to the database engine and is also not very time consuming, having only a linear complexity of $\mathcal{O}(n * m)$, where n is the total number of snapshots and m the amount of metrics for which the trend line should be calculated (*V.R2*).

Aggregate Function Importance of aggregated values differ depending on the type of aggregate function used, but also on the metric to analyze. Regarding different aggregates, experts were more interested in median values, while average values were considered as least important. To provide a wide variety of analytic capabilities, the prototype offers three different types of aggregate functions from which the user can choose, though not all aggregates are supported by every metric (*V.R2*, *V.R10*). The available options are:

- **Sum** Cumulative metrics can be summed up over all artifacts for which they were recorded. This feature is currently only implemented for *Source Lines of Code*, but adding additional metrics like total amount of bugs or similar could be a useful extension at a later stage. The cumulative value is currently determined by summing up all SLOC metric values per file artifact of the corresponding snapshot, but also here, additional artifact filters for classes or methods could be conceivable for further versions. Although the significance of this metric, when viewed in isolation, is relatively limited, when combined with another metric it could provide an orientation for the overall size of the project (*V.R3*).
- **Average and Median** These two options let the user choose between the aggregated values that should be printed in the trend diagram (*V.R2*). The aggregates are calculated over all recorded artifacts, a breakdown on artifact level (class, method or file) is currently not supported. Although *median* was the preferred option by most experts, adding *average* values did not require much implementation overhead, as all aggregate functions are already provided by the ArangoDB API[26] and were precalculated during the mining phase, switching between the different options hence does not require any additional computing time.

Min- and Max Values In addition to the calculated aggregates, the lowest and highest metric values that were recorded for any artifact within a snapshot can also be displayed if required.

Uniform Axis Scaling In order to improve comparability between individual projects, this option can be used to adjust the Y-axis scaling of all diagrams to a common

[26] `https://www.arangodb.com/docs/stable/aql/functions-numeric.html`, last accessed on 28.05.2020

value range. For this, the lowest and highest values of all measured aggregated values across all projects are selected and form the new lower and upper bound of all trend diagrams (*V.R3*, *V.R9*). See also Figure 6.6 for more details.

Figure 6.6: Different trend diagrams without (top) and with (bottom) uniform scaling applied. By using a common y-axis scaling, it becomes obvious that the value ranges of the two outmost projects on the left side are partly a way below the two on the right side, a fact that would not immediately apparent if every diagram uses its own axis scaling.

Thresholds This topic is highly related to the concept of health indicators, and will therefore be considered separately in the corresponding section 6.4.2.

This dashboard view allows the quality experts to quickly gather a rough overview of the entire portfolio and it also provides possibilities to compare individual projects within the portfolio directly with each other. However, depending on the portfolio size, there is still a large amount of information encoded in the metric trends, which has to be extracted manually in order to reason about the overall portfolio's quality state. To support the quality expert with this task, the dashboard view was extended with additional health indicators (7). These indicators are subject of the next section.

6.4.2 Health Indicator

In addition to the analysis of portfolio metrics over time, the continuous monitoring of its health state also plays an important role for quality managers (see requirements *V.R7* and *V.R8*). The prototype therefore defines the concept of a *health indicator*. Such an indicator is composed of several *rules*, where each rule defines the (optional) lower and upper boundaries between which a specific metric value should be. As soon as a metric value moves below or above a corresponding threshold, the rule is considered as violated and the health state of the portfolio is changed accordingly. There is also an additional tolerance range before each actual threshold: If the metric value is within this area, it is not regarded as incorrect per se, however, the trend development of this metric will be monitored to recognize any possible decline of the portfolio's software quality. In that way, quality engineers can gain a fast overview of the portfolio's health trend without analyzing the different metric trends manually. Since in general, not the entire course of the portfolio is of interest, but rather only the recent development (and here in particular a looming trend, if available), only the last n snapshots are used for determining the overall health status, whereby this parameter is configurable. Furthermore, in order to

avoid false-positives or outliers, a rule violation is only counted as such if at least 50% of the last n snapshots are above or below the thresholds. Figure 6.7 should describe this behavior in more detail: The diagram shows an arbitrary metric trend that was recorded over the last snapshots. The dotted vertical line on the right side defines the area of interest when determining a possible rule violation, as only metric values right of this line will be considered. The thresholds that are defined by a rule are represented by the red lines, the given rule will be considered as violated if more than half of the values within the interest area are below or above these lines. A small area below (or above) each threshold (currently 10% of the distance between the origin and the lowest threshold value) define the monitoring area that will be used when determining the metric trend. While in this concrete diagram, the last metric value is clearly within the monitoring area, the majority of the relevant metric values still lies in the optimal area and therefore the overall state, at least when regarding this rule, is considered as *healthy*. Here, the last value is interpreted as an outlier, but the situation could change if future values might also be in the monitoring area.

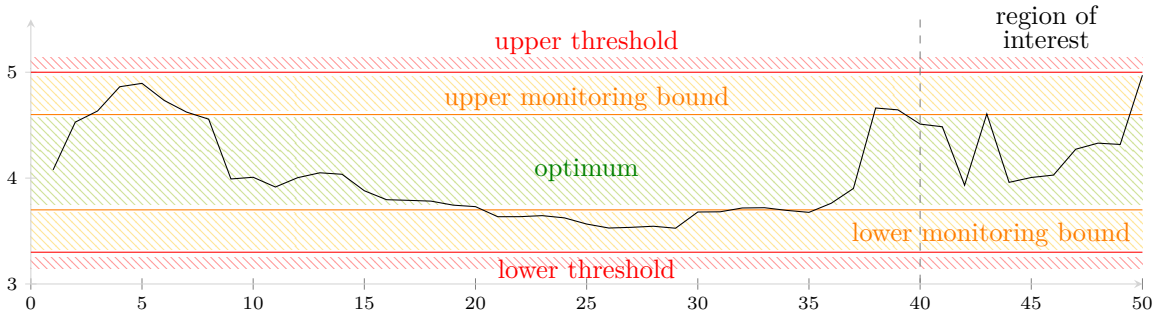


Figure 6.7: Different threshold areas when monitoring metric trends

If the majority of the metric values are located within a monitoring area (either in the lower or the upper area), the development of the trend curve will be further investigated to determine the direction of the metric trend. This is done in a two step process: (1) to avoid any possible distortion because of outliers, the metric trend curve is smoothed by calculating a moving average and (2) by calculating a straight line between the first and last relevant metric values, the gradient of the trend is calculated. In case of the upper bound, a negative gradient would mean that the trend is moving away from the threshold, therefore the portfolio's health state is improving. A positive gradient on the other side would indicate that the trend is moving towards the critical area and countermeasures should be taken to avoid any threshold violation. Figure 6.8 illustrates this approach: A straight line is drawn from the beginning of the area of interest to the most recent metric value and by solving the linear equation $y = mx + n$ for these two points, the gradient of the line can be determined.

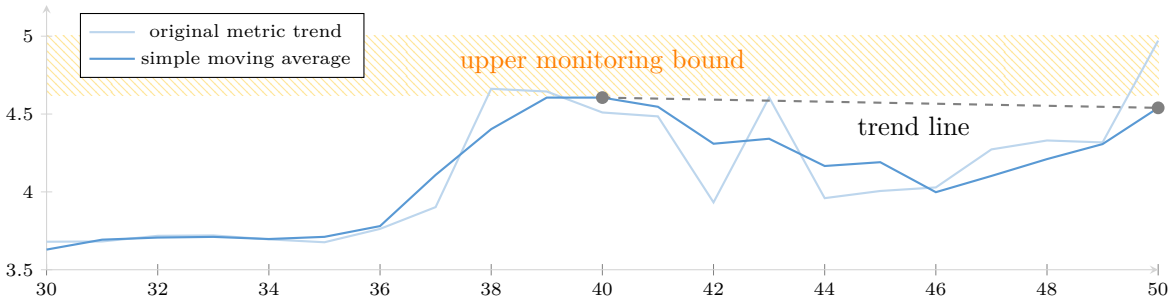


Figure 6.8: Determining the direction of the metric trend

Based on this procedure, a health indicator can have five different states:

- **Not calculated** Threshold violations have not been calculated so far. This can happen if either a new metric rule was added to the indicator or an existing rule was changed. In that case, the recalculation of the rule must be triggered manually. This state can also be reached in case that no metric rules were defined at all.
- **All metrics within parameters** All metric values of the relevant snapshots are within the optimal parameter range.
- **Trend is improving** This state is reached whenever the recorded metric values are approaching any thresholds, but without yet violating them. In case of an upper bound, the trend would move away downwards from the threshold, in case of a lower bound, the trend would move upward.
- **Trend is getting worse** The opposite situation of the previous case: Thresholds are not yet violated, but the trend is clearly moving towards them. In that case, quality managers should pay attention on how the metric values will develop in the future.
- **Metric thresholds are violated** One or more rules are currently violated and appropriate actions should be taken accordingly.

The different health conditions are ordered by their severity, that means as soon as a rule enters the *Critical* state (meaning that the threshold of the rule is violated), the overall health of the portfolio will also become *Critical*, even if there would be other rules with a better state. This behavior underlines the alarming-character of the indicators and also reduces the information load here significantly (*V.R7*, *V.R4*): Quality managers get a notification as soon as the first problem occurs, and all other information becomes irrelevant as long as this problem exists.

Editing indicators and defining rules can be achieved through a modal dialog that pops up when the user clicks on an arbitrary health indicator symbol in the dashboard view (see Figure 6.9). The dialog gives an overview of all rules that are currently defined for

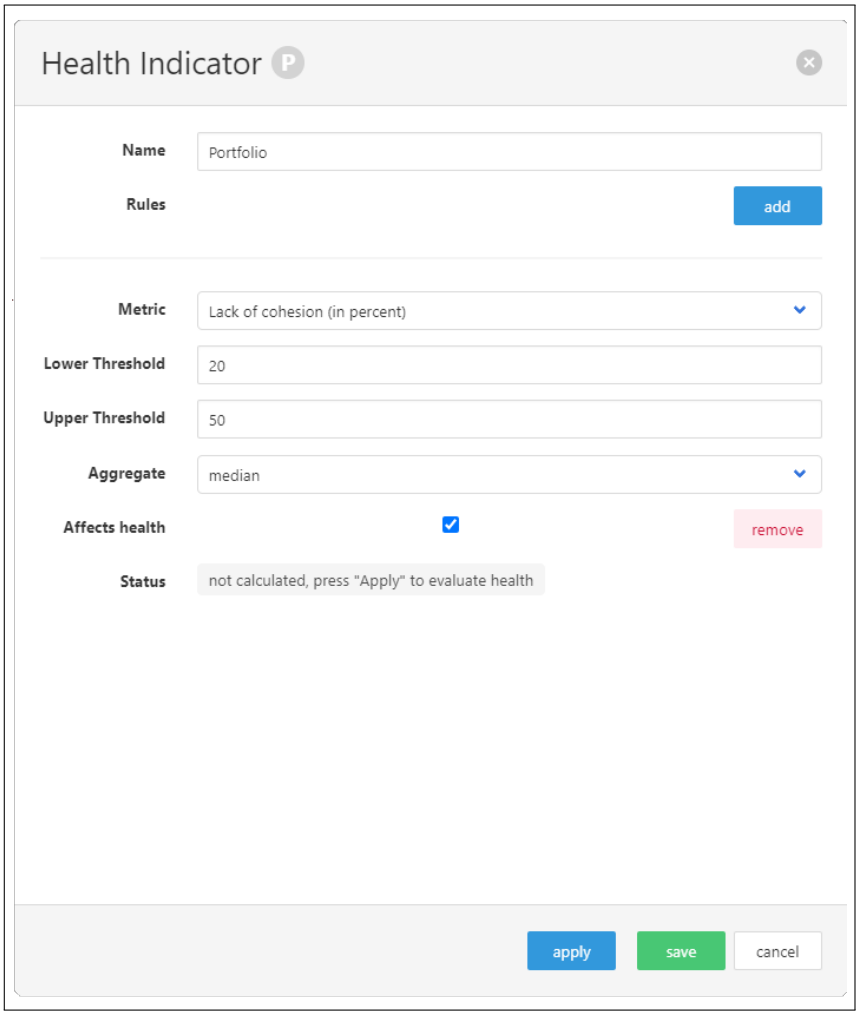


Figure 6.9: Modal dialog for editing the rules of a health indicator

the selected indicator. When creating a new rule, the user can choose from all available metrics in the system and selecting the type of aggregate value to which this rule should be applied (Average, Median, Sum, Min or Max value) is also possible. To trigger a recalculation of the portfolio's health state after creating or changing a rule, the *Apply* or *Save* button has to be pressed. It can also be set for each rule individually, whether its violation affects the portfolio's overall health state. This option can be useful, for instance, for prototypes or early development phases, where the quality expert already wants to track the condition of the project, but without fearing negative impact on the overall portfolio. As soon as the development advances further or the prototype finally goes into production, the corresponding rule could then be activated and the project's health state would then also directly affect the portfolio health.

Some experts mentioned during the interviews that, due to the possible inhomogeneity

of a portfolio, globally defined thresholds are often not meaningful enough to adequately reflect the health status of a portfolio. There can be projects within the portfolio that could be security critical or might express more complex domain logic and therefore also need to be evaluated separately. To support this need, the prototype also offers the possibility to define indicators individually per project. If such a project indicator is defined, its rules will be applied instead of the ones of the global indicator to validate the project's health. Figure 6.10 illustrates this feature in more detail:

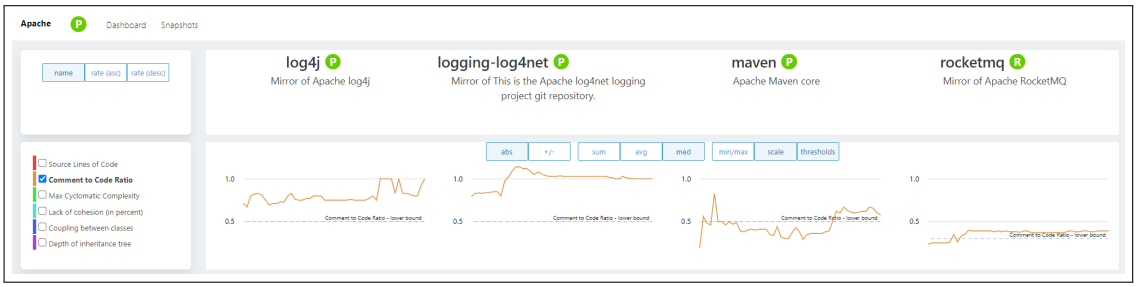


Figure 6.10: While three of the projects use the global portfolio indicator (P), the RocketMQ project uses its own custom indicator (R) with a lower metric threshold.

The first three projects use the global portfolio indicator as can be seen by the letter *P* in the middle of the health indicator circle both next to the portfolio name and the relevant projects. This indicator defines a metric rule with a lower *Comment To Code Ratio* threshold of 0.5, placing the metric values of the first three projects in the optimal area. The last project, RocketMQ on the other hand, has a comment ratio that is much lower than 0.5 and would therefore be in critical condition regarding this rule, but since it uses its own project indicator (note the *R* letter in its health circle) with a metric threshold of 0.3, the project health stays positive and therefore also the overall portfolio health.

6.4.3 Snapshot View

The dashboard view is helpful for getting a quick overview of the portfolio, but when it comes to a deeper analysis of trends and their causes, the information display reaches a certain limit. Especially switching to a more detailed artifact level (*V.R6*) requires a different visualization approach than simply combining several trend series within a single line chart, as this quickly become impractical because of an increasing information density. Therefore, in addition to this main view, there is also the option to switch to another snapshot-centered view via the menu bar. This view (see Figure 6.11) can roughly be divided into two different, vertically arranged components: The upper screen area presents a trend diagram and a slider control that can be used for navigating through the various historical project trends to select a specific snapshot. The lower part of the screen is composed of different detailed views, with which different aspects of the selected snapshots can be analyzed. The different parts of the view should be described shortly in the following section:

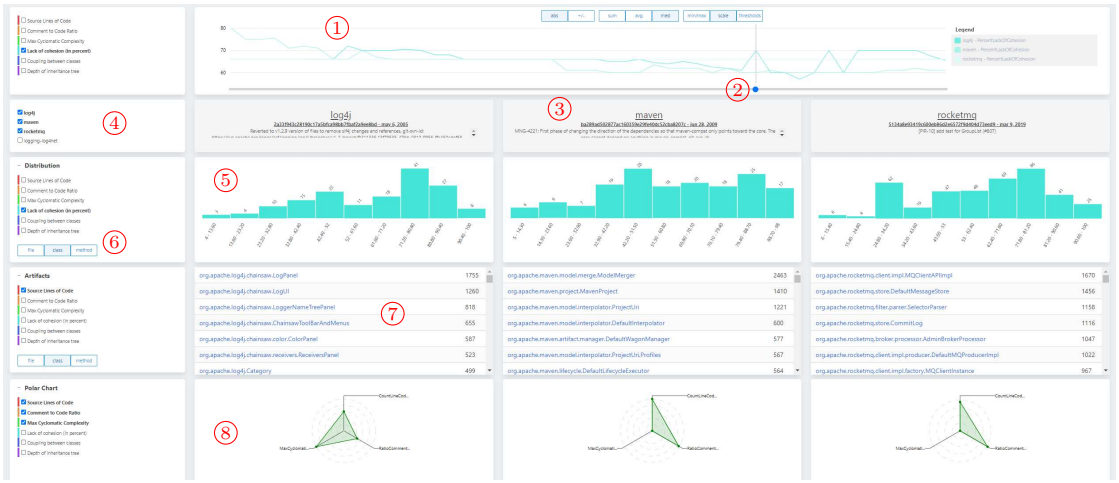


Figure 6.11: Snapshot centric view with following elements: (1) combined trend graph for selected projects, (2) slider to navigate to individual snapshots, (3) meta data of selected snapshots, (4) list of projects selected for visualization, (5) distribution of selected metric for current snapshot, (6) artifact filter, (7) list of metric values per artifact, (8) polar chart with normalized metric values

Navigation Area

In contrast to the trend diagrams used in the portfolio dashboard, here the historical trends of several projects can deliberately be combined into a single line chart to be able to directly compare the metrics of different repositories more easily (1). However, the user must be aware that this can lead to a possible information overload and should therefore keep the number of selected projects and metrics as low as possible. By using a slider control placed at the bottom of the diagram (2), the quality expert can navigate through the trend curves and thus select a concrete snapshot for further analysis. As with the portfolio view, the scaling of the X-axis is based on the total number of sampled snapshots, so if the slider is placed exactly in the middle of the diagram, it points exactly to the 25th snapshot of each project (assuming a sample number of 50 snapshots per project). However, depending on the development life cycles of the various projects, the selected snapshots can sometimes be far apart in time. Also analogous to the portfolio dashboard, various aggregate values and overlays, such as min/max metric values or custom thresholds, can also be displayed here, but with the difference that all data will be rendered within a single diagram.

Snapshot Meta Data

After choosing a snapshot index with the navigation slider, the meta data of the corresponding snapshots are displayed in a column-oriented bar located right below the trend diagram, where each selected repository is represented by one column (3). The shown data include hereby the Git commit hash, the date of the commit and the commit

message. The background color of the snapshot's expresses the health state of the project at the specific time, in case the user has defined any metric thresholds via a health indicator. Since only a single snapshot is considered here, an analysis of the metric trend, as done in the portfolio dashboard, is not available. Instead, in this view the health state are only expressed by four different colors used as background color for the snapshot panels.

- No health state was calculated for this snapshot. This state can also be reached if no thresholds were defined.
- Metric values are in the optimal range at the time of the selected commit
- Metric values are close to the lower or upper limit of the valid metric range. However, since this is a only a single snapshot, no trend-related information is available.
- One or more metric values are violated during this commit.

If a rule's threshold is almost reached or even violated, a list of the relevant metric names is displayed in addition to the snapshot data. In order to establish a direct connection between the analysis and the underlying repository's source code (see *V.R11*) the snapshot's tile also contains a HTML hyperlink which points directly to the corresponding commit website on Github.

Distribution

The next panel (Figure 6.12) uses histograms to visualize the distribution of metric values among the various source code artifacts (5). To limit the information density, the user can filter the artifacts by choosing one of three available artifact types (6), which are in descending order of their granularity level: *files*, *classes* or *methods*. For calculating the histograms, the range of values per metric was split up into different buckets (also called bins), and for each bin, the number of metric values that fall within its range were counted. Every metric uses custom and equal value ranges for their bins, while the last bin normally covers all remaining values. Since collecting distributions can be relatively time-consuming, this calculation is already done during the mining phase and the results are stored in a separate document collection. Requesting the relevant histogram data during the visualization can then be achieved through a simple database query without any additional computational overhead.

Artifact List

In addition to the previously introduced distribution view, the individually recorded metric values can also be displayed separately in a source code artifact list (7). This list contains an entry for each examined code artifact and its determined metric value, in descending order. Every entry is also directly linked to its Github repository page via a hyperlink, allowing the user to jump directly to the corresponding source code if

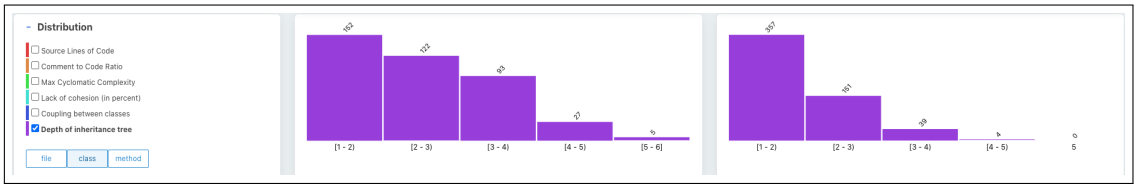


Figure 6.12: Histograms are used to visualize the distribution of selected metric values per snapshot.

necessary. Compared to the portfolio dashboard, this list view thus provides the most granular access level to information within the portfolio, as with its help, users can trace metric values directly back to their originating source code artifact as demanded by requirements *V.R6* and *V.R11*. Similar to the histograms, this view can also be filtered by three different artifact types.

Polar Chart

Finally, the last panel allows the combination of several metric values into a single polar chart (8) as also shown in Figure 6.13. Currently, all metrics are visualized by their median value (or sum value for cumulative metrics) with the corresponding value being normalized on its entire value range occurring in the project. For example, if the cyclomatic complexity metric value for this snapshot is in the middle of its axis, this means that the corresponding value corresponds to half of the maximum measured median value for this metric. In this way, it can be determined how quickly one metric value increases or decreases compared to other metrics and thus, possible patterns could be recognized more easily.

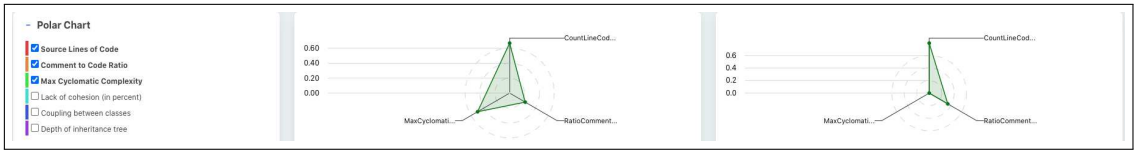


Figure 6.13: While in the first diagram, the cyclomatic complexity has reached its peek already after approx. 60 % of the lines of code have been written, the second diagram shows a project where the maximum lines of code are almost reached, but the cyclomatic complexity is still considerably low.

6.4.4 Summary

The visual prototype was realized in accordance with the knowledge gained from the analysis of various visual concepts. As the research has shown, combining all relevant data into a single, holistic view would probably be unsuitable to implement all of the requirements. Such a view would, on the one hand provide too much information at

once, resulting in information overload and overplotting and on the other hand, the different visualization aspects and various levels of detail would also be difficult to express throughout a single view. Instead, the prototype uses two different main views providing different visualization concepts and detail levels.

The first view, the portfolio dashboard, provides a general overview of all projects within the portfolio and lets the user also configure health indicators to monitor the health state of the whole portfolio and its containing projects. The other screen, the snapshot-centric overview provides a finer level of information granularity and lets the user select a specific point within the historical trend data and focus her or his analysis specifically on this point. For this, the prototype provides different visualizations, like histograms, polar charts and also list views containing source code artifacts together with their metric values. Hyperlinks are used to trace the collected metric values directly back to the underlying source code. To prevent any potential information overload, all implemented visualization solutions provide various filter and customization options.

The views implemented so far are certainly only a small selection of possible visualization options and many other diagram or chart types would also be very conceivable, but had to be omitted due to time constraints. Nevertheless, adding and evaluating more visualization options could be an interesting point for further research in this area.

After implementing the prototype, the next consequent step had to be its practical evaluation. For this, some real-world scenarios were developed to test the practicability of the individual features with the help of an expert group. This evaluation process will be addressed in the next chapter.

CHAPTER 7

Evaluation

For finding answers to the remaining research questions and also for the final assessment of the visualization system, a scenario-based expert evaluation was carried out, similar to the *Technical Action Research* approach described in [24]. The goal of this procedure is to test the quality and functionality of an experimental artifact, in this case the visualization prototype, under the most realistic conditions possible. For this, a test plan consisting of seven different scenarios was developed, each one reflecting a different task in the area of software quality management and analysis. Furthermore, a group of six experts in the area of software development and quality management was selected as the test audience which try to solve the aforementioned tasks with the help of the visualization prototype. The results of this test execution were documented accordingly and then used for further evaluation.

The following chapter provides an overview of the evaluation process, beginning with a description of the test plan, together with an explanation of the test scenarios. After a brief presentation of the actual test execution follows the evaluation and interpretation of of the collected answers.

7.1 Test Plan

According to [35], three different test phases can basically be distinguished: (1) *Explorative Tests* are mainly carried out during the early project phases to identify the specific user's and customer's needs and to define the corresponding requirements and concepts. (2) Based on these findings, *Assessments or Summative Tests* are then executed at a later development stage. While the main goal of these tests is to evaluate the usefulness and general usability of the implemented features, this work also uses the participants' answers here to draw conclusions regarding the previously defined questions. Tests in this phase are best defined as scenarios which require the users to use the prototype to solve a particular real world problem. During the test execution, any intervention by

the moderators should be kept to a minimum as far as possible to not influence the test subjects. (3) Finally, *Validation or Verification Tests* are used to evaluate the prototype against defined benchmarks prior to release [35].

In the present work, the exploratory phase was essentially already covered by a systematic literature search followed by various expert interviews (see section 2.3), therefore an additional exploratory test phase was dispensed with. Instead, the test plan was primarily based on summative and verification tests.

Besides the test scenarios developed for the summative tests, the survey contained also additional questions which purpose was to gather further insights into the experiences the users gained while dealing with the prototype. The test plan thus consists of the following areas:

Demographic Data There are some demographic factors that might affect the visual perception of a person to a greater or lesser extent [80], as well as other criteria, such as the test person's underlying experience, could also have a possible impact on the test result. Even if the total number of participants in this study is probably too small to make conclusive statements regarding demographic influences, the collection of this data can nevertheless help in gaining interesting insights. Due to the small test group size and the qualitative collection of data through one-on-one discussions with the participants, it is difficult to collect the demographic data in a completely anonymous way, especially since the limited sample size makes it theoretically possible to assign several demographic factors (like gender and age) to a specific individual. Therefore, submission of these data is purely optional and the consent of the participants for the collection of this demographic data was explicitly obtained beforehand. In the context of this survey, the following demographic data was collected for each participant:

- Age
- Gender
- Experience as software developer in years
- Experience in years regarding work that is only related to analyzing and managing software quality.
- Share of software quality-related daily work (in percent) in comparison of one's total daily workload

Test Scenarios The different test scenarios will be described in more detail in section 7.2

Research Questions An essential goal of the evaluation was also to answer the research questions previously defined in chapter 1.3. While some of these questions could already been answered during earlier phases of this work, the scenario-based test cases aim specifically at answering questions *RQ3* to *RQ5* (assessment of the

visual and functional capabilities of the prototype). In order to collect answers for questions *RQ3* and *RQ4*, the test plan was expanded by two further, final questions:

Q1: *Did the visual interface of the prototype support you adequately in solving the corresponding tasks or do you think the interface was not well suited for these tasks* (derived from question *RQ3*)?

Q2: *How would you rate the amount of information presented* (derived from question *RQ4*)?

Both questions can be answered based on a Likert scale [27], additional space for optional free text notes by the participants is also available for each question. Research question *RQ5*, which is about the overall evaluation of the prototype by the experts, on the other hand, could only be answered comprehensively after all results of the surveys were evaluated and will therefore be discussed separately in section 7.5.

System Usability Scale Evaluating the effectiveness and usability of any type of software system with a visual interface is not a trivial task and a sophisticated evaluation can often require a large amount of resources, therefore the System Usability Scale (SUS) was developed as a more lightweight alternative [36]. This scale consists of ten questions, whose answers can be used to assess the global usability of a system based on subjective perceptions of the users. The possible answers are organized on a five-digit Likert scale, reaching from 1 (*strongly disagree*) to 5 (*strongly agree*). The complete scale can be found in the appendix (see 15).

7.2 Scenarios

Seven test cases were constructed for the actual scenarios, each one based as closely as possible on real-world use cases. Every scenario consists of a short task description and additional background information to put the scenario into a wider context. Additionally, a success criterion representing the expected solution was also defined for each scenario, but was hidden from the participants.

In order to be able to qualitatively evaluate the functionality of the prototype, the participants were also asked to rate the difficulty of the given task on a Likert scale, assuming that they would only use existing tools that were familiar to them. In a first version of the test plan, the users were also asked to briefly describe the steps that were necessary to complete the tasks with their existing tools, but during the course of the pilot evaluation, it turned out that answering these questions only from memory can be quite difficult, so it was decided to skip this step and only use the aforementioned Likert scale. The experts should also give a rough estimation of how much time they think it would take them to complete the task based on their chosen tools. These estimates were later used as a benchmark value by comparing it with the actual time spent on solving

the tasks with the visual prototype. However, besides that, the estimated time is also a possible way to guess how well an existing tool would be able to solve the given task. High estimates could indicate a lack of functionality by the given tool, which would be necessary to solve the scenario. Although temporal estimations may be subject to greater fluctuations and should therefore be treated with caution, this procedure might still be able to provide sufficiently sound results, especially since letting the participants execute all test scenarios with their own tool chain would require a considerable amount of time and resources.

To ensure the validity of the test results [81], the participants were also asked to assess the relevance of the respective test scenario using a Likert scale. This is to later identify irrelevant scenarios and minimize their effect on the overall result.

The chosen scenarios were hereby:

Scenario I: Comparing metric trends

Description	Regarding Maven, do the changes of the average Lines of Code metric correlate stronger with the changes of the average Comment to Code Ratio or with the average Cyclomatic Complexity metric?
Background	When comparing metrics with a different value range, it can often be better to compare their relative changes instead to find possible relations between the trends.
Solution Steps	1) Open Portfolio Dashboard 2) Select metrics *Lines of Code*, *Documentation to Code Ratio* and *Cyclomatic Complexity* in one diagram. Change the aggregate function to *average*. 3) Change control panel setting from *Absolute* to *Relative*
Success Criteria	The Complexity metric trend should clearly correlate with the SLOC metric.

Scenario II: Determining the health state of the portfolio and its projects on lower detail level

Description	Regarding their latest trend, which projects have a median lack of cohesion (LCOM) that is higher than 65?
Background	The *Single Responsible Principle* suggests that classes should at best only have one single purpose. The *Lack of Cohesion* metric can be used to measure this feature. During this scenario, it should be tested how the tool could support users by setting quality constraints and monitoring them.
Solution Steps	1) Open Portfolio Dashboard

2) Click on the portfolio's health indicator in the top left corner.
3) Remove any existing metric rules if present
4) Add a new metric rule for the *LCOM* metric and set the upper threshold to 65. Choose *median* as aggregate function.
5) Press *Apply* or *Save* to confirm the changes.
6) Click the *Rate (asc)* button to sort the projects by their health state.

Success Criteria After sorting, log4j and RocketMQ are violating the threshold, while log4net and maven have either decreasing or increasing trends.

Scenario III: Determining the health state of the portfolio and its projects on higher detail level

Description Regarding its latest trend, which project has an artifact (file or class) that has a maximum cyclomatic complexity of more than 50?

Background Metric thresholds cannot only be applied on project level, but also on a more fine-grained artifact level, like on files or classes. This scenario evaluates the possibility of the tool to monitor metrics on a very fine detail level by still keeping an overview of the whole portfolio.

Solution Steps
1) Open Portfolio Dashboard
2) Click on the portfolio's health indicator in the top left corner.
3) Remove any existing metric rules if present
4) Add a new metric rule for the *CC* metric and set the upper threshold to 50. Choose *max* as aggregate function.
5) Press *Apply* or *Save* to confirm the changes.
6) Click the *Rate (asc)* button to sort the projects by their health state.

Success Criteria After sorting, RocketMQ should be the only project with a critical health state.

Scenario IV: Define individual health criteria per project

Description Keep the portfolio state to green even if RocketMQ does violate the previously defined cyclomatic complexity rule.

Background Portfolios can contain different projects with various complexities and requirements, it should therefore be possible to evaluate

their health state individually if necessary. There could also exist prototypes or project in early development, which, while still be monitored, should not affect the overall health of the portfolio.

Solution Steps

1) Open Portfolio Dashboard
2) Click on the project health indicator of the RocketMQ project.
3) Choose the *Create a new indicator for this project* option.
4) Either change the metric threshold to a higher, more suitable value or uncheck the *Affects health* checkbos for the rule.
5) Press *Apply* or *Save* to confirm the changes.
6) Check the portfolio's health state again.

Success Criteria The portfolio's health state should change to green, while the RocketMQ health state should still be red.

Scenario V: Identify single classes within the whole portfolio

Description Regarding Maven and RocketMQ, which is the highest class coupling value a class has and which class is it?

Background If a class has too many couplings, it can be difficult to maintain the class as all the dependencies to other classes have to be kept in mind. A class with high couplings could therefore be a possible subject for a refactoring. This scenario checks the possibility the tool provides to navigate from a bird's eye portfolio view into the details of a single class.

Solution Steps

1) Open Snapshot Overview
2) Select all four projects in the top trend diagram.
3) Activate the *Class Coupling* metric.
4) Active the *Min/Max* overlay in the line chart.
5) Find the highest maximum value by manually checking the overlay and using the slider to verify the correct value (the maximum number is shown as tooltip when the mouse is hovered over the diagram).
6) Choose one of the last snapshots where the maximum value should be 137.
7) Expand the artifact list
8) Select the *Class Coupling* metric from the metric list
9) Choose the *class* filter from the filter list

Success Criteria The class MQClientApiImpl from the RocketMQ project has the highest coupling.

Scenario VI: Comparing projects on different detail levels

Description	What might be a possible reason that the total lines of code of log4j and log4net became more equal?
Background	Monitoring and comparing the evolution of two projects can be useful when making project related decisions. This scenario therefore evaluates the possibility to combine cross project trend analysis on different levels.
Solution Steps	1) Open Snapshot Overview 2) Select the log4j and log4net projects. 3) Activate the *Lines of Code* metric. 4) Compare the slider positions between Feb 10, 2005 and May 6, 2005 (log4j) 5) Expand the artifact list 6) Select the *Lines of Code* metric from the metric list 7) Choose the *file* filter from the filter list 8) Identify the files with the highest SLOC count and open the files by clicking on their external link.
Success Criteria	Some UI related files like LogPanel.java or LogUI.java (for Chainsaw maybe?) are missing in the later commit. Maybe additional UI support was either dropped or moved to a different repository.

Scenario VII: Monitoring metrics over time

Description	Regarding the Maven project, how does the amount of classes with one base class and classes with two base classes change between May 15, 2009 and Jun 20, 2020?
Background	Monitoring the trend of metrics over time within a single project can provide meaningful insights about the evolution of a project. This scenario demonstrates how information can be derived from the charts, although there is no visual component which visualizes the information directly. Instead, users should be able to use the visualization as a foundation for their further analysis.
Solution Steps	1) Open Snapshot Overview 2) Select the Maven project. 3) Expand the Distribution list. 4) Select the *Depth of Inheritance Tree* metric from the metric list and choose the *class* filter. 5) Move the slider between the two relevant dates.

6) Write down the numbers of the first two bins at the corresponding slider positions.

Success Criteria At the first date, there should be 288 elements in the first bin and 77 elements in the second bin. During the second date, these numbers should change to 407 and 173.

7.3 Preparation

Before carrying out the actual tests, the necessary hardware and software components were checked for their operability during a pre-test phase. Also, as suggested by authors like Rubin and Chisnell [35], a pilot phase was scheduled where both, the scenarios and the general testing procedure were evaluated. The following two sections should give a brief presentation of this pre-test phase, starting with the hard-and software setup and concluding with the insights that were gained through the pilot test.

7.3.1 Setup

Because of contact restrictions due to the Covid 19 pandemic[1], the evaluation was carried out exclusively online through video conferences with desktop sharing. For this, three different video conference applications (Jitsi [2], Microsoft Teams[3] and Zoom[4]) were tested on two different systems, a Lenovo ThinkPad with Debian 10.4 Buster and a MacBook Pro Mid 2012 with macOS Catalina 10.15.4 installed.

Among the examined systems, Zoom performed best, as the other systems either had problems when sharing the screen or providing remote access. Microsoft Teams even crashed unexpectedly under Debian when trying to activate desktop sharing due to a configuration issue[5].

A web browser was then hosted on the test system with three (during the pilot phase: two) browser tabs, containing the following content:

1. The web front end of the prototype application hosted through a docker container (see below).

2. The Google Forms[6] online questionnaire which has been tailored for each specific user (see section 7.4).

[1] `https://www.who.int/emergencies/diseases/novel-coronavirus-2019`, last accessed on 11.07.2020

[2] `https://meet.jit.si/`, last accessed on 11.07.2020

[3] `https://www.microsoft.com/en-us/microsoft-365/microsoft-teams/group-chat-software`, last accessed on 11.07.2020

[4] `https://zoom.us/`, last accessed on 11.07.2020

[5] `https://techcommunity.microsoft.com/t5/microsoft-teams/debian-10-teams-share-screen-blake-crash/m-p/1068428`, last accessed on 11.07.2020

[6] `https://www.google.com/forms/about/`, last accessed on 11.07.2020

3. After the pilot-phase: A handout describing the user interface and functionality of the prototype as a PDF document (see Appendix 15).

The prototype itself was started on the test computer with the help of three orchestrated Docker[7] containers. One container was hosting the ArangoDB database, another one provided the Python backend and the third container was responsible for starting the web server that hosted the web front end. The test participants were given remote access rights to the test machine and its browser window through an initially established Zoom session.

Based on this setup, a pilot test was carried out to refine the questionnaire and the selected scenarios for the later test execution [35].

7.3.2 Pilot Test

For the pilot phase, a participant was chosen who broadly corresponds to the target audience. In this case the attendee was a 39 year old, male application developer with approx. 16 to 20 years experience in general software development and six to ten years experience in quality related tasks. Regarding the percentage of quality-related daily tasks, however, the test person stated only about 10% of their total working hours, which was, compared to the numbers mentioned by the target group later (see section 7.5.1), at the bottom of the scale.

The test person received a brief oral introduction to the functioning of the prototype and the structure of the test plan and was then asked to fill out the questionnaire and to solve the tasks by using the prototype. During the execution, the test moderators took notes and recorded the time the subject required to solve the tasks. The results of the scenario evaluation can be read from Table 7.1.

Overall, the pilot tester got along relatively well with the prototype, the time required for solving the various scenarios was either less or within the previously estimated time periods. It should be mentioned here, however, that some tasks required additional assistance from the moderators (see improvement list below). Apart from scenario VI, the tasks were also consistently considered *relevant* to *very relevant*. Regarding the research questions *Q1* and *Q2*, the test person stated that the prototype was well suited to solve the required tasks and that the amount of information presented was also very manageable. The *System Usability Scale* survey further showed that the system was generally easy to use (again, see suggestions for improvement below) and that most of the functions were well integrated and not unnecessarily complex.

The pilot test also revealed some shortcomings that should be addressed prior to the actual test phase:

[7] `https://www.docker.com/`, last accessed on 11.07.2020

Scenario	Recorded time with prototype	Estimated time with existing tools	Relevance 1 - irrelevant 5 - very relevant
I	11 min	10 - 30 min	⋆⋆⋆
II	8 - 11 min	10 - 30 min	⋆⋆⋆⋆⋆
III	2 min	5 - 10 min	⋆⋆⋆⋆⋆
IV	50 - 60 sec	n/a	⋆⋆⋆⋆⋆
V	6 - 8 min	10 - 30 min	⋆⋆⋆⋆
VI	2 - 4 min	5 - 10 min	⋆⋆
VII	5 min	30 min - 60 min	⋆⋆⋆

Table 7.1: Evaluation of pilot test phase

- **Ambiguity** Some scenarios were not formulated unambiguous enough and left too much room for interpretation, not only in regard of the result but also in the tasks itself. The relevant scenarios were therefore reformulated to be more explicit.
- **Availability of Features** Creating health indicators by clicking on the indicator symbol next to the portfolio or project name was not immediately obvious to the test person, as in this case the UI does not follow common patterns (the element was not recognized instantly as clickable). While this feature was briefly introduced during the initial presentation, providing the participants with a detailed handout or cheat-sheet (see Appendix 15) would be much better, as then they would not have to memorize all features during the introduction. In the long term, a more recognizable user interface concept for editing health indicators might be an even better solution.
- **Axis description** The test subject criticized also the lack of axis labels. Since adding additional axis titles would have required some significant changes regarding the row layout that could not have been implemented at short notice, the labeling of the axes was also adopted in the handout.
- **Contrast** The lack of contrast within the chart diagrams (especially when activating the min/max overlay) was also mentioned by the pilot tester, this effect was even partially reinforced by the remote display. Investigating this issue revealed that this effect could not be easily remedied without making major changes to the color palette. Instead, the selection of projects to be examined was reduced in some scenarios, which, thanks to a reduced number of color shades, made it easier to distinguish between the individual metric trends.
- **Scrolling** Due to hardware restrictions, the screen resolution for the pilot phase was chosen quite small which lead to undesired scrolling of the view. This problem

was solved by using an external monitor with a higher resolution during the test phase.

- **Questions that are difficult to answer** Remembering the different execution steps to solve a task revealed to be quite difficult and the accuracy regarding the concrete order and number of steps might also be debatable. Therefore, the relevant questions were dropped and instead replaced by a Likert scale used to evaluate the overall complexity of each scenario when executed with existing tools.

7.4 Test Execution

For the actual evaluation, six members of a research and development company with approximately 300 employees were chosen to form the target audience group, whereby all six participants were employed as software developers/architects at their institute. To counteract any possible influences resulting from the fact that three participants had already taken part in the expert interviews for gathering the prototype's requirements, three additional persons who were not involved in the prototype development were selected as control group. The exact demographic distribution of the participants as well as a categorization of their knowledge and experience level is described in more detail in section 7.5.1.

According to [82], the outcome of a test scenario could be affected by its position within a sequence and the number of scenarios preceding and following it. To counterbalance these learning effects, the mentioned survey suggest a procedure that can be used to organize the test scenarios in a Latin square in such a way that the ordering of the test cases varies for each participant. Constructing a Latin square with the algorithm described by Bradley [82] requires an even number of test scenarios, so applying this method on the given test plan, consisting of seven scenarios, was not possible without modifications. But since scenario IV highly depends on its predecessor, they were both considered as a single scenario, which ultimately resulted in an even number of scenarios for creating the final test sequences. Table 7.2 gives an overview of the individual test sequences: The rows indicate the respective participant (A-F), the columns in turn determine the position (1-6) of the scenario (I-VII) within the individual test run.

For the test interviews, each participant was invited to a Zoom video conference, together with two moderators. The test case execution via video conference with additional remote access almost worked all of the time without any problems, there was only one situation where the remote access could not be handed over to the user as planned, but this was solved by changing the operating system on the participant's side.

Individual/Order	1	2	3	4	5	6
A	I	II	III+IV	V	VI	VII
B	III+IV	I	VI	II	VII	V
C	VI	III+IV	VII	I	V	II
D	VII	VI	V	III+IV	II	I
E	V	VII	II	VI	I	III+IV
F	II	V	I	VII	III+IV	VI

Table 7.2: Test Scenarios organized in a Latin Square. For every participant (A-F), each test scenario (I-VII) has a different predecessor and successor.

Based on [81], each test was separated in the following four stages:

Preparation During this phase, which was normally completed before the participant joined the meeting, the prototype's Docker container was started up and the relevant information was loaded into the web browser. The initial page of the online questionnaire was loaded into the browser as was the first page of the cheat sheet.

Introduction As soon as the test subject joined the meeting, she or he was welcomed by the moderators and the desktop, where the test environment was running, was shared with all members of the interview. The interviewee then received a brief introduction to the background and the planned course of the test. As part of this introduction, the questionnaire and the cheat sheet were also briefly presented to the participant.

Execution After the introduction, the participant gained remote access to the test environment and was asked to proceed with the test. This execution stage could roughly be separated into six different steps:

1. First, the participant was asked to fill out the demographic questions of the online survey.
2. After that, but before starting with the actual scenarios, the participant was given sufficient time to consult the handout. The corresponding browser tab stayed open during the whole test, so the interviewee could come back to the cheat sheet whenever desired.
3. At the beginning of each scenario, the participant was then asked to answer the initial questions regarding the severity and the required time to solve the task with existing tools.
4. Now the actual problem solving stage started: The participant switched to the browser tab with the prototype front end and tried to solve the task with the help of the application. During this process, the moderators were available at

any time to answer questions and provide support if needed. The moderators only gave explicit tips on how to solve the problem if it was necessary due to usability difficulties - for instance, if the user was not able to figure out a specific feature even after consulting the cheat sheet or in case the application did not work as expected. The time it took the participant to solve the task was measured and written down separately by both moderators, as were any additional feedback or comments that were mentioned by the interviewee during the test.

5. After completing the task, the test subject was also asked to rate the relevance of the scenario she or he just solved.
6. As soon as all seven scenarios were completed, the participant had to answer the two remaining research questions (*Q1* and *Q2*) and conclude the questionnaire with a SUS survey [83], which also finished the execution phase of the test.

Debriefing Following the execution stage, an open discussion was held with the participant about his impressions and experiences in handling the prototype, and possible findings were also recorded here by the two moderators.

The average time for processing all four stages per participant was between 60 to 90 minutes, leading to an overall time for all appointments of about six to eight hours. During the execution stage, almost all participants were practicing a method known as *thinking out aloud* [81], meaning they were paraphrasing and verbalizing their actions and thought process while interacting with the prototype. This allowed the moderators to get a qualitative understanding of the participants' experience while using the tool. It should also be noted that, due to the prototypical but nevertheless complex nature of the application, the moderators supported the participants with additional hints if those were stuck during a scenario. In order to avoid preferring individual participants through this, however, it was always ensured that all participants received roughly the same amount of assistance.

7.5 Result

7.5.1 Demographics

Five of the six participants stated their gender as male, one as female. The average age of the interviewed experts was 34 years with all members of the test group between 31 and 40 years.

As can be seen in Figure 7.1, the experience in general software development stated by the participants correlated strongly with their age: People under 33 years of age stated 6-10 years of experience, participants between 33 and 34 chose the next higher level (11-15 years) and people in the age group of 35 and older stated their experience as 16 to 20 or more than 20 years. Regarding their specific experience in software quality management and analysis, the groups were shaped much more homogeneous: Here all

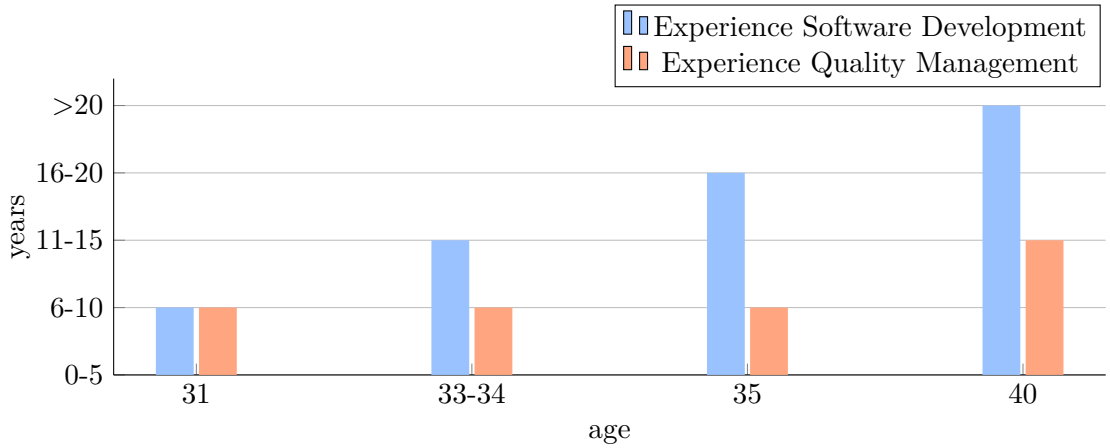


Figure 7.1: Experience of the test audience grouped by age. While the experience in general software development grew with the participants' age, the stated experience in software quality management remained constant except for the oldest age group.

participants chose the category 6-10 years, except for the oldest, who chose 11-15 years. Based on the ratio of age to experience, the test group therefore corresponds exactly to the pilot candidate.

In contrast, the test group gave a higher number on average when asked about the ratio between quality related tasks and their daily work: While the pilot participant's answer was with 10% at the lower end of the scale, the test audience had an average ratio of 18%.

7.5.2 Scenario Evaluation

The evaluation of the individual scenarios is mainly based on two aspects: Firstly, the qualitative responses given by the participants were used to identify possible advantages or disadvantages of the prototype, and secondly, the estimated severity and time required to solve the scenarios (assuming that only existing and known tools are used) were compared to the actually measured time that was needed to complete the task by using only the prototype (see Figure 7.2 for an overview of the measured times per scenario). In that way, a relatively accurate assessment of the prototype's performance, compared to existing solutions, could be made.

Scenario I was assessed differently by the participants, both in terms of severity and in terms of the estimated time required. Two experts estimated the effort to be relatively high (30-60 minutes), but the majority chose a time frame of five to ten minutes. Compared to these estimates, the average time to solve the task with the prototype was 3:29 minutes, with only one case where finding the solution took longer than five minutes (6:29). Most participants realized quite early that comparing the absolute trends does not lead to the desired result due to the different value ranges of the selected metrics and changed the

diagram visualization option to *relative*. After that, identifying the relevant trends was relatively straight forward. While the scenario was initially developed for the dashboard view, some testers also used the single line chart in the snapshot view - both options lead to the expected result.

Scenario II was rated as only slightly more difficult and most testers estimated 5 to 30 minutes to solve the task. Interestingly, here the participants seemed to get along better with the prototype, on average they only needed 2:30 minutes to complete the scenario, although the spread was much smaller here than in the previous challenge. There were some interviewees that tried to solve the scenario by simply comparing the trend graphs manually, in which case the moderators suggested the use of a health indicator. All participants were able to create and configure a global health indicator easily, while there was a certain uncertainty whether the threshold should be defined as lower or upper bound. Eventually, most participants chose the correct option.

Scenarios III and **IV**, which build on one another, were rated equally complex and were both classified as more difficult than the previous ones. However, this assumption was not reflected in the estimated effort: Here the participants seemed to be at odds: While half of the test audience stated they would only need less than five minutes for scenario III, the other half estimated the effort to be ten to 30 minutes. In contrast, the effort for scenario IV was mostly estimated between five to ten minutes, with one estimate with less than five minutes and one with more than 30 minutes. The prototype did quite well here in comparison, finding a solution for scenario III took 3:24 on average, for the following scenario IV, however, only another 1:29 minutes. The interviewees' approaches for this task were mostly similar to the one used for scenario II and creating the global indicator worked in most cases without any problems. Defining the project-related indicator for scenario IV was a bit more challenging, as some participants were no longer aware how these individual indicators could be created, which partly endorsed the findings from the pilot phase that this feature should have been designed in a more intuitive way. For instance, one user suggested a direct navigation between the global and individual indicators, which could greatly improve the usability here.

The assessment of **Scenario V** shows a not entirely unified picture: Most of the interviewees stated that they could solve the scenario in less than five minutes, but two also stated that it would take them between 10 to 30 and 30 to 60 minutes. In comparison, solving the challenge with the prototype took the participants an average of 3:15 minutes, with two cases taking over four minutes. However, it must be taken into account that some participants mentioned that they had difficulties in identifying the correct solution because of the low color contrast due to the remote desktop sharing.

The two **Scenarios VI** and **VII** were rated as the most difficult ones, accordingly, the processing time for scenario VI was also estimated by most experts with 30 to 60 minutes. This might be related to the fact that most interviewees were unsure whether they would be able to find the correct reason of the projects' divergences at all by just analyzing the source code. Scenario VII was classified as even more difficult, but here the effort estimation experienced a quite large fluctuation: All answers from five minutes to more

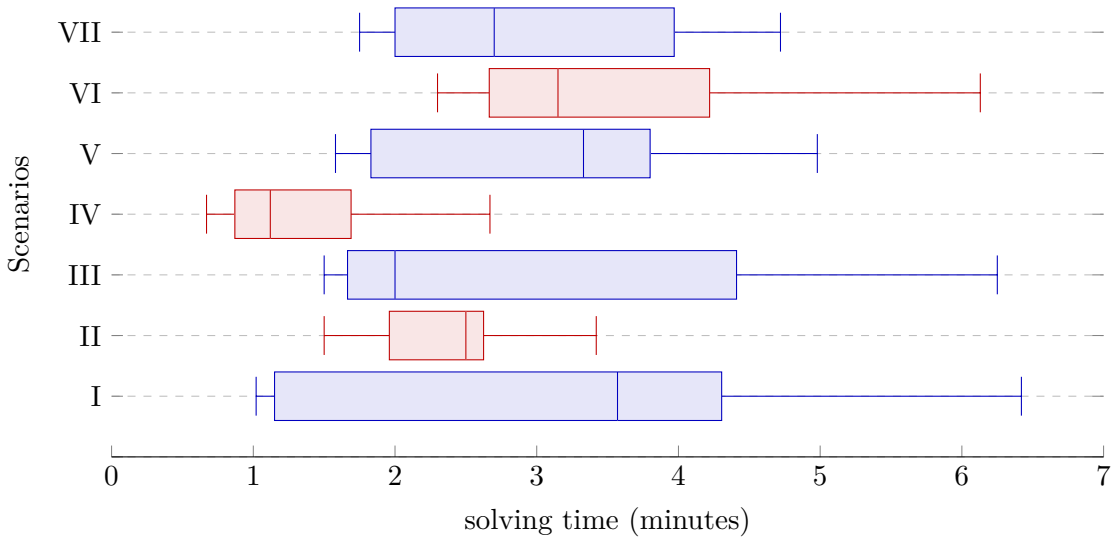


Figure 7.2: Scenario Solving Time: While there were some outliers, most of the scenarios could be solved in a relatively short amount of time by the experts, especially the ones that were previously estimated as quite difficult to solve (VI and VII).

than an hour were represented. The actual processing time for both scenarios were 3:42 (VI) and 3:50 minutes (VII) and were therefore only slightly higher than for scenarios I, III and V. Especially for these two tasks, some participants asked for an additional diff function to compare the code and metrics of two commits more easily. The possibility to view the distribution of different metrics was very well received by most experts, as they considered this as a feature that is not supported by most other commercial tools.

Overall, the results of the expert evaluation were quite promising: In a direct comparison, the processing times for solving the scenarios with the prototype were at least as good or better than the initial expert estimates when using existing tools. Only in scenarios III and V there seemed to be a tie: At least half of the participants stated that they could solve the scenario in less than five minutes, which corresponds to the measured times of 3:24 respectively 3:15 minutes. In this area, the prototype would therefore compete relatively strongly with existing solutions, at least at project level. But regarding scenarios that were assessed as relatively difficult (VI and VII), it has also been shown that here the processing times were only slightly higher than for the other tasks and, in some cases, were even far below the estimates. This could indicate that these scenarios represent some type of problem that is rather difficult to solve with existing tools. In this particular area the prototype could offer a real added value for software quality managers.

With regard to the relevance of the scenarios, the participants largely agreed that the selected tasks reflected real-world challenges relatively well: All scenarios were classified in the range of *average relevant* to *very relevant* and none was marked as *not* or *less*

relevant. Scenario II was rated the most relevant by the testers, closely followed by scenario IV. The relevance of the latter might be due to some participant's focus on working on single projects instead of managing whole software portfolios, therefore setting individual quality gates on a per project base is an important feature for these experts. Scenario VI was considered the least relevant by most of the participants. Some noted that there might exist specific use cases where the direct comparison of source code and metrics between two projects might be relevant, but since most projects are usually quite inhomogeneous, gaining any additional insights based solely on these comparisons can be very difficult.

7.5.3 Research Questions

In response to the question of how well the individual functions were implemented within the prototype (Question *Q1*), most of the participants gave four out of five possible points, whereas one participant rated the application as sometimes too complex and very difficult to use. In particular, the options for editing the health indicators on both, global and project level, were not immediately apparent to the participants. Some also mentioned they would prefer further display options here to see which rules were active and which are directly responsible for the current portfolio status (for instance, in the form of additional badges in front of the indicator symbols). Overall, the functional range of the prototype was highlighted as positive by most of the participants, as some of the offered features were not available in existing applications, such as the visualization and comparison of various metrics and their trends across project boundaries. The possibility to freely combine metrics within one diagram and to choose different aggregate functions was also noted as an advantage over existing solutions. In some cases, the participants even wanted more extensive visualizations, such as the representation of the trend curve of individual distributions or the further combination of different graphs within a single diagram.

When it comes to evaluating the amount of information shown (Question *Q2*), four of the six participants considered the data quantity as either easy or very easy to handle, whereas two participants noted that the application was sometimes confusing due to the large amount of data displayed. The dashboard in particular was often described as being too cluttered, some users suggested to add additional configuration options for adding and removing rows individually, which could reduce the information density in this area significantly. There was also one person who expressed the experience that while the amount of data seemed a bit overwhelming to him at first glance, he quickly learned to get along and focus on the relevant parts as the test progressed. As a further option for information reduction, two participants also suggested adding more filter options to the artifact list, like limiting the visible artifacts to a specific area like a certain package or directory. A zoom function within the diagram would also be welcomed by some users, as it could especially be useful if one wants to investigate the metric trend between two commits in more detail. However, this would also require a higher commit sample rate as the current one that was used during the repository mining. One participant also

raised his concerns that with increasing scaling (e.g. more projects or more metrics) the information density, especially in the dashboard area, would also increase accordingly. Additional configuration options, like choosing only a subset of all available metrics or projects for the final visualization, could counteract the then appearing information overload.

7.5.4 System Usability Scale

Five of the six experts that participated in the evaluation rated the prototype with more than 70 points in the SUS survey, with one participant seeing the usability somewhat more critically and assigning only 47.5 points. The main points of criticism were the already mentioned information density, which was considered as to be too high occasionally, together with some user interface concepts, such as the selection of health indicators, which were found to be unintuitive. In addition, some more usability related features, such as the lack of contrast regarding the buttons, but also in some diagrams, were also criticized. On average, the individual evaluations resulted in an average total score of 76.67 points (see Figure 7.3), which, on the supplementary SUS adjective scale, would roughly be equivalent to an "excellent", assuming that the highest value of the previous category *good* would reach till 75 [84]. Overall, the survey shows a very positive picture of the prototype, even if there is certainly room for improvement in some areas.

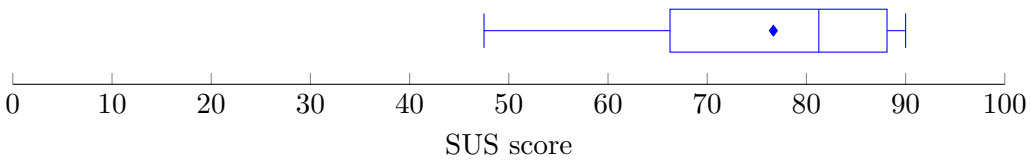


Figure 7.3: Result of the SUS survey. The average score of the prototype was 76,67 points on the System Usability Scale.

7.6 Summary

From an overall perspective, the prototype was well received by the test audience and many participants found their way around very quickly, although occasional help from the moderators was necessary in some situations.

Comparing the real execution time with the experts' estimates, in combination with the qualitative survey, seemed to be a sufficiently accurate approach to assess the prototype's performance. However, it should be kept in mind that this type of qualitative survey does not scale very well for large user groups due to the amount of work involved in gathering and analyzing the data. Based on the measured values, the prototype performed in the majority of cases just as well or better, compared to existing solutions. The fact that some scenarios, which were initially rated as relatively difficult by the experts, could be solved relatively quickly later on by using the prototype also reveals that there might

exist some kind of niche that is not yet sufficiently covered by existing quality tools. This could be the area were the prototype would create real benefit for software quality managers. But on the other hand, there were also use cases where the prototype was mostly on par with existing tools. This was especially the case for scenarios related to individual project analysis, which on the other hand, were not the main focus of this prototype.

7.7 Discussion

The evaluation of the questionnaire provided sufficient information to finally answer the main question of the work, „How would an expert visualization system have to be designed to support software quality managers in analyzing software portfolios?“, together with the corresponding sub-questions. While question *RQ3* about the concrete requirements for the expert prototype, was essentially answered in Chapter 6 already, the qualitative interviews did also show that most participants were satisfied with the functional scope of the prototype, so most of the previously identified requirements could be considered as relevant. The answers to the research questions about the functionality *RQ3* and information density *RQ4* could directly be derived from the responses given by the participants to the survey questions *Q1* and *Q2*. Here, the evaluation showed that the previously set research goals could mostly be achieved: According to the experts, the chosen visualization concepts turned out to be quite suitable for illustrating the collected data and solving the given challenges. The same applies, albeit to a somewhat more limited extent, to the information density of the expert system: Most users stated that the amount of visible information was well manageable most of the time and that a possible information overload did either not occur at all or only in rare cases. However, there were also two users who found the systems or at least parts of it as overly complex and difficult to handle, even when considering the complex nature of the underlying data. Some participants mentioned that especially the information density in the dashboard could further be reduced by adding additional configuration options. Taking the qualitative feedback and the results of the SUS survey into account, it can savely be assumed that the prototype was mostly met with wide acceptance by the participants, thus providing an answer for research question *RQ5*. Returning to the main question, it can therefore be said that the previously defined requirements were correctly identified and also successfully implemented by the prototype. In addition, the experts' answers also allow the conclusion that the tool provides real added practical value for quality experts, that goes beyond pure academic interests.

7.8 Threats to Validity

According to [85], there exist three major validity areas in any research approach that can be affected by various threats, and any of these three must be addressed and handled separately:

Construct validity describes how well the measured results are suited to represent the context that appears in the real world. For instance, this research uses the time that is required to solve a task as a benchmark to evaluate the performance of the prototype versus existing tools. While time along might not be a sufficient factor to draw any conclusions, here it is only used in combination with other indicators like the estimated task complexity and also qualitative interviews. All these factors together should help in reducing any possible threat in this area.

Internal validity can be threated by unknown factors that affect the results without being recognized. In the context of this evaluation, some of these threats that had to be considered were:

- A possible learning effect when executing the scenarios always in the same order. This effect could be minimized by providing every participant with a different order of scenarios (see section 7.4).
- A biased test audience. Some of the participants were also involved in previous surveys connected to the prototype which could have affected their answers. To compensate this effect, an additional control group of three experts were added to the test audience. Nevertheless, it must be noted that all expert were working for the same institute and therefore cannot be considered as a representative sample.
- Any visual or performance related issues due to the remote video connection. To minimize any influence on the results here, participants were asked to ignore any usability related issues where possible.

External validity deals with the question how well the results can be applied to larger groups. Due to the small sample size, the results of this study cannot be regarded as representative for the whole target audience of quality managers, especially since the participants mentioned they on average spend only 18% of their time on quality related issues. But nevertheless, the survey allowed for finding answers to the research questions and to evaluate the prototype, based on the subjective impressions and experiences of some selected experts in this area. Carrying out a representative research would have required a much larger amount of test subjects, which would have clearly exceeded the scope of this work.

CHAPTER 8

Conclusion

Software portfolios are becoming more and more important, be it due to the growing emergence of microservice architectures or the increased usage of various, in parts even very polyglott, software solutions within today's companies. Based on this trend, an increasing need for quality management and analysis within these portfolios can be derived. The conducted research within the scope of this work revealed that, although such a genuine need for software quality management on portfolio level exists, so far this area has not been adequately covered, neither from an academic perspective, nor when regarding available applications and tools (see chapter 3).

Therefore, this diploma thesis aimed at filling this gap by answering the question about how an expert system would have to be designed to support quality experts on portfolio level. This was achieved by creating and evaluating a prototypical expert system.

To establish a solid foundation for further analysis, the thesis first answered the research questions *RQ1* and *RQ2* about quality experts' information needs and how these needs would be manifested in concrete requirements. With systematic literature research and additional qualitative expert interviews, a set of unique information needs could be identified that differ from those related to single project analysis. The answers to these questions can be rated as an essential contribution of this work to this area since, so far, the information needs on portfolio level have not been analyzed and identified to this extent.

In the next step, a set of requirements was defined that directly addressed these information needs, and the relevance of these requirements could also be acknowledged through qualitative expert interviews. Therefore, the work described in Section 6.2 could very well be considered a requirement catalog that further portfolio analysis tools should at least consider to satisfy. Questions *RQ3* and *RQ4*, which ask for concrete visualization concepts and how these could be used to implement the aforementioned requirements, could also be answered adequately. A selection of visual concepts, together with their

advantages and disadvantages, and the application of these concepts, are presented in Chapters 2 and 6. A final expert evaluation could confirm the quality and suitability of the used concepts. In the course of this evaluation, also an answer to the last research question about the acceptance rate of the prototype among experts could be found. The experts' feedback were very positive and revealed that the prototype was very well suited to solve the given tasks. Especially the holistic approach, covering both project and portfolio level visualization, and the various configuration options to individually adjust the information density were positively highlighted, as none of the existing tools provide such a sophisticated level and combination of visualization features on portfolio level. In this area the prototype could be viewed as a very innovative novelty and could also help as a possible reference implementation or starting point for further research.

In addition to this visual expert system, the thesis also dealed with the subject of how a possible data acquisition process had to be defined to collect the metric data relevant for the visualization. As an answer to research question *RQ6*, it could be shown that such a process could operate at a very high automation level, without any human interaction except for some inital configuration steps. The mining toolchain was also developed with the idea in mind to support a variety of different metric and repository providers as requested by questions *RQ7* and *RQ8*. It could also be shown here that the implementation of such a system is possible and external components could easily be plugged in by implementing a specific interface. This level of customizability is also quite new, compared to other data mining processes investigated in the course of this work.

8.1 Further work

Quality management and analysis of software portfolios is still a very broad research field and even if the contributions of this diploma thesis are only a very small part of the overall research, they could still serve as a starting point for further, promising studies. To conclude, some further research ideas, more or less based on this underlying work, should therefore be briefly presented.

Broader Analysis of Information Needs
When it comes to the specific information needs of quality managers at portfolio level, research activity is still relatively low: There exist smaller studies that deal with this particular area, like [2], but a large-scale survey, as the one that was carried out at project level by Buse and Zimmermann [15], is not yet available on portfolio scale. The survey that was carried out as part of this thesis (Section 6.1) can only be regarded as an initial contribution to this area, as due to the small amount of participants, the results cannot be seen as representative for the whole group of software quality managers. A larger survey would certainly be able to provide some other important insights in this area, but would also require much more resources. In addition, most investigations so far focused on questioning quality managers from commercial software companies, but interviewing experts from other types of organizations, like open source foundations or communities, would certainly also be interesting, as quality managers there might have

different information needs and requirements.

Integration of Additional Metrics
In its current form, the mining tool chain only supports an exemplary selection of metrics, all of which can be determined by using only static code analysis. However, as as mentioned by some experts during the surveys, there is also a high interest in metrics like, for instance, the test coverage, which can only be determined when the compiled code is available. This would require an additional compilation step during the mining process, which would certainly increase the overall process runtime significantly, so further attempts to optimize and parallelize the workflow would be necessary.

More Visualization and Configuration Options
Not all suggested requirements and features that were identified in the analysis found their way into the prototype - for instance, the automatic hot spot detection would certainly be an interesting feature to implement, as it was also suggested by some experts during the evaluation phase. Besides that, the participants of the survey also mentioned some other interesting functions, like additional filters which could help in temporarily excluding unrelated artifacts from the analysis, or an option to zoom into the trend graph. Additional configuration options (to reduce the amount of graphs in the dashboard for example) could also reduce the information density further. In addition, another often requested feature during the evaluation was a diff function that would allow for viewing the differences between metrics and artifacts between two given snapshots. Apart from these configuration options, it would also be interesting to introduce further visualization concepts, like a trend graph showing the evolution of individual distributions or a bubble chart such as those used in SonarQube - to only name a few.

Additional Artifact Tracking
Today's software development is not only focused on plain source code, there are many other program artifacts that should be part of a holistic quality management approach. Some of these artifacts could be database schemes, environment configuration for DevOps processes or event Machine Learning models are important artifacts that evolve during the development process and should therefore be also tracked and analyzed. Depending on the type of artifact, a complete new set of metrics or visualization concepts might be necessary.

Integration into Continuous Development Processes
Quality analysis seldom stands on its own, but must generally always be seen in the context of a continuous development and release process. Nowadays, these processes are highly automated, although mostly only on project level. An extension of these processes to entire portfolios, together with adequate quality analysis, is a field that offers great potential for further research.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.

[2] Suvi Elonen and Karlos A. Artto. „Problems in managing internal development projects in multi-project environments“. In: *International Journal of Project Management* 21.6 (2003), pp. 395–402. ISSN: 02637863. DOI: 10.1016/S0263-7863(02)00097-2. URL: http://dx.doi.org/10.1016/S0263-7863(02)00097-2.

[3] Daniel Simon, Kai Fischbach, and Detlef Schoder. „Application portfolio management–an integrated framework and a software tool evaluation approach“. In: *Communications of the Association for Information Systems* 26.1 (2010), p. 3.

[4] Jarno Vähäniitty, Kristian Rautiainen, and Casper Lassenius. „Small software organizations need explicit project portfolio management“. In: *IBM Journal of Research and Development* 54.2 (2010). ISSN: 00188646. DOI: 10.1147/JRD.2009.2038747.

[5] Nuha Alshuqayran, Nour Ali, and Roger Evans. „A systematic mapping study in microservice architecture“. In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2016, pp. 44–51.

[6] Jan Bosch and Petra Bosch-Sijtsema. „From integration to composition: On the impact of software product lines, global development and ecosystems“. In: *Journal of Systems and Software* 83.1 (2010), pp. 67–76. ISSN: 01641212. DOI: 10.1016/j.jss.2009.06.051.

[7] Irakli Nadareishvili et al. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[8] Don Coleman et al. „Using Metrics to Evaluate Software System Maintainability“. In: *Computer* 27.8 (1994), pp. 44–49. ISSN: 00189162. DOI: 10.1109/2.303623.

[9] Jarrett Rosenberg. „Some misconceptions about lines of code“. In: *International Software Metrics Symposium, Proceedings* (1997), pp. 137–142. ISSN: 09205489. DOI: 10.1016/s0920-5489(99)92169-4.

[10] Christian R. Prause et al. „Is 100% test coverage a reasonable requirement? Lessons learned from a space software project“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10611 LNCS (2017), pp. 351–367. ISSN: 16113349. DOI: 10.1007/978-3-319-69926-4_25.

[11] Henrike Barkmann, Rüdiger Lincke, and Welf Löwe. „Quantitative evaluation of software quality metrics in open-source projects“. In: *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*. 2009, pp. 1067–1072. ISBN: 9780769536392. DOI: 10.1109/WAINA.2009.190.

[12] Kecia A.M. Ferreira et al. „Identifying thresholds for object-oriented software metrics“. In: *Journal of Systems and Software* 85.2 (2012), pp. 244–257. ISSN: 01641212. DOI: 10.1016/j.jss.2011.05.044. URL: http://dx.doi.org/10.1016/j.jss.2011.05.044.

[13] Steffen M Olbrich, Daniela S Cruzes, and Dag I.K. Sjøberg. „Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems“. In: *IEEE International Conference on Software Maintenance, ICSM*. September 2010. 2010. ISBN: 9781424486298. DOI: 10.1109/ICSM.2010.5609564.

[14] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. „Comparing software metrics tools“. In: May 2014 (2008), p. 131. DOI: 10.1145/1390630.1390648.

[15] Raymond P L Buse and Thomas Zimmermann. „Information needs for software development analytics“. In: *Proceedings of the 2012 International Conference on Software Engineering*. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 987–996. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337343.

[16] Tobias Kuipers and Joost Visser. „A Tool-based Methodology for Software Portfolio Monitoring“. In: (2011), pp. 118–127. DOI: 10.5220/0002682301180127.

[17] S. Komi-Sirviö, P. Parviainen, and J. Ronkainen. „Measurement automation: Methodological background and practical solutions - A multiple case study“. In: *International Software Metrics Symposium, Proceedings* (2001), pp. 306–316. DOI: 10.1109/metric.2001.915538.

[18] K B Akhilesh and K. B. Akhilesh. *Portfolio Management*. 2014, pp. 185–191. ISBN: 9786468600. DOI: 10.1007/978-81-322-1946-0_14.

[19] Andy Siddaway. „What is a systematic literature review and how do I do one“. In: *University of Stirling* I (2014), p. 1.

[20] Michael Balzer, Oliver Deussen, and Claus Lewerentz. „Voronoi treemaps for the visualization of software metrics“. In: (2005), p. 165. DOI: 10.1145/1056018.1056041.

[21] D Moher et al. „Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement (Reprinted from Annals of Internal Medicine)“. In: *Physical Therapy* 89.9 (2009), pp. 873–880. ISSN: 1549-1676. DOI: 10.1371/journal.pmed.1000097.

[22] Shyam R. Chidamber and Chris F. Kemerer. „Towards a metrics suite for object oriented design“. In: *ACM SIGPLAN Notices* 26.11 (1991), pp. 197–211. ISSN: 15581160. DOI: 10.1145/118014.117970.

[23] S Keshav ACM SIGCOMM Computer Communication Review and Undefined 2007. „How to read a paper General knowledge“. In: *Computing.Dcu.Ie* (2012), pp. 1–3. URL: http://www.computing.dcu.ie/{~}ray/teaching/CA485/notes/01{_}how{_}to{_}read{_}a{_}paper.pdf.

[24] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[25] David J. Flinders. „InterViews: An introduction to qualitative research interviewing“. In: *Evaluation and Program Planning* 20.3 (1997), pp. 287–288. ISSN: 01497189. DOI: 10.1016/s0149-7189(97)89858-8.

[26] Barbara A Kitchenham and Shari L Pfleeger. „Personal opinion surveys“. In: *Guide to advanced empirical software engineering*. Springer, 2008, pp. 63–92.

[27] Robert Wall Emerson. „Likert Scales“. In: *Journal of Visual Impairment & Blindness* 111.5 (2017), pp. 488–488. ISSN: 0145-482X. DOI: 10.1177/0145482x1711100511.

[28] Virginia Braun and Victoria Clarke. *Thematic analysis, APA Handbook of Research Methods in Psychology*. 2012. DOI: 10.1037/13620-004.

[29] Thomas Epping. *Kanban für die Softwareentwicklung*. Springer-Verlag, 2011.

[30] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[31] Russ Miles and Kim Hamilton. *Learning UML 2.0*. " O'Reilly Media, Inc.", 2006.

[32] Robert C Martin. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[33] David Janzen and Hossein Saiedian. „Test-driven development concepts, taxonomy, and future direction“. In: *Computer* 38.9 (2005), pp. 43–50.

[34] Roy Osherove. *The Art of Unit Testing: With Examples in. Net*. Manning Publications Co., 2009.

[35] Jeffrey Rubin and Dana Chisnell. *Handbook of usability testing: how to plan, design and conduct effective tests*. John Wiley & Sons, 2008.

[36] John Brooke. „SUS: a “quick and dirty'usability“. In: *Usability evaluation in industry* (1996), p. 189.

[37] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. „MetricMiner: Supporting researchers in mining software repositories“. In: *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013* September (2013), pp. 142–146. DOI: 10.1109/SCAM.2013.6648195.

[38] G. Robles. „GlueTheos: automating the retrieval and analysis of data from publicly available software repositories“. In: (2006), pp. 28–31. DOI: 10.1049/ic:20040471.

[39] Martin Auer, Bernhard Graser, and Stefan Biffl. „An approach to visualizing empirical software project portfolio data using multidimensional scaling“. In: *Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration, IRI 2003* (2003), pp. 504–512. DOI: 10.1109/IRI.2003.1251458.

[40] M. Ohira. „Empirical project monitor: a tool for mining multiple project data“. In: (2006), pp. 42–46. DOI: 10.1049/ic:20040474.

[41] Sakamoto Yasutaka et al. „Visualizing software metrics with service-oriented mining software repository for reviewing personal process“. In: *SNPD 2013 - 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (2013), pp. 549–554. DOI: 10.1109/SNPD.2013.96.

[42] Shinsuke Matsumoto and Masahide Nakamura. „Service oriented framework for Mining Software Repository“. In: *Proceedings - Joint Conference of the 21st International Workshop on Software Measurement, IWSM 2011 and the 6th International Conference on Software Process and Product Measurement, MENSURA 2011* (2011), pp. 13–19. DOI: 10.1109/IWSM-MENSURA.2011.28.

[43] Di Wu et al. „A metrics-based comparative study on object-oriented programming languages“. In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE* 2015-Janua (2015), pp. 272–277. ISSN: 23259086. DOI: 10.18293/SEKE2015-064.

[44] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[45] W Pree. „Essential Framework Design Patterns“. In: *Object Magazine* 7 (1997), pp. 34–37. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.6880{\&}rep=rep1{\&}type=pdf.

[46] Diomidis Spinellis. „Tool writing: A forgotten art?“ In: *IEEE Software* 22.4 (2005), pp. 9–11. ISSN: 07407459. DOI: 10.1109/MS.2005.111.

[47] Lh Rosenberg and Le Hyatt. „Software quality metrics for object-oriented environments“. In: *Crosstalk Journal, April* 10.4 (1997), pp. 1–6. URL: http://people.ucalgary.ca/{~}far/Lectures/SENG421/PDF/oocross.pdf.

[48] Thomas J McCabe. „A complexity measure“. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[49] Engineer Bainomugisha et al. „A survey on reactive programming“. In: *ACM Computing Surveys* 45.4 (2013). ISSN: 03600300. DOI: 10.1145/2501654.2501666.

[50] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[51] Khaled El-Emam. „Object-Oriented Metrics: A Review of Theory and Practice“. In: *Advances in Software Engineering* March (2002), pp. 23–50. DOI: 10.1007/978-0-387-21599-0_2.

[52] Bonnie A. Nardi. „The use of scenarios in design“. In: *ACM SIGCHI Bulletin* 24.4 (1992), pp. 13–14. ISSN: 07366906. DOI: 10.1145/142167.142171.

[53] Eirini Kalliamvakou et al. „The promises and perils of mining GitHub“. In: *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings* (2014), pp. 92–101. DOI: 10.1145/2597073.2597074.

[54] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. „PyDriller: Python framework for mining software repositories“. In: *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 908–911. DOI: 10.1145/3236024.3264598.

[55] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. „Mining software repositories with iSPARQL and a software evolution ontology“. In: *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*. 2007. ISBN: 076952950X. DOI: 10.1109/MSR.2007.21.

[56] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall New York, 1988.

[57] Jan Palach. *Parallel Programming with Python*. 2014, p. 124. ISBN: 9781783288397.

[58] Donald E. Knuth. „Computer Programming as an Art“. In: *Communications of the ACM* 17.12 (1974), pp. 667–673. ISSN: 15577317. DOI: 10.1145/361604.361612.

[59] Robert H.B. Netzer and Barton P. Miller. „What Are Race Conditions?: Some Issues and Formalizations“. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.1 (1992), pp. 74–88. ISSN: 15577384. DOI: 10.1145/130616.130623.

[60] D. N. Card and C. L. Jones. „Status report: Practical software measurement“. In: *Proceedings - International Conference on Quality Software* 2003-Janua.July (2003), pp. 315–320. ISSN: 15506002. DOI: 10.1109/QSIC.2003.1319116.

[61] Siniscalco Maria Teresa and Aurat Nadia. „Questionnaire design: Module 8“. In: *Quantitative research methods in educational planning* (2005), pp. 22–35. ISSN: 0011-3921. DOI: 10.1177/001139219804600403. arXiv: arXiv:1011.1669v3.

[62] Holger M. Kienle and Hausi A. Müller. „Requirements of software visualization tools: A literature survey“. In: *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* July 2007 (2007), pp. 2–9. DOI: 10.1109/VISSOF.2007.4290693.

[63] Miroslaw Staron et al. „Dashboards for Continuous Monitoring of Quality for Software Product under Development“. In: *Relating System Quality and Software Architecture* (2014), pp. 209–229. DOI: 10.1016/B978-0-12-417009-4.00008-9.

[64] Daniel A. Keim. „Information visualization and visual data mining“. In: *IEEE Transactions on Visualization and Computer Graphics* 8.1 (2002), pp. 1–8. ISSN: 10772626. DOI: 10.1109/2945.981847.

[65] Robert Grant. *Data visualization: Charts, maps, and interactive graphics*. 2019, pp. 1–249. ISBN: 9781138707603.

[66] Michele Lanza and Stéphane Ducasse. „Polymetric views - A lightweight visual approach to reverse engineering“. In: *IEEE Transactions on Software Engineering* 29.9 (2003), pp. 782–795. ISSN: 00985589. DOI: 10.1109/TSE.2003.1232284.

[67] Martin Pinzger et al. „Visualizing multiple evolution metrics“. In: *Proceedings SoftVis '05 - ACM Symposium on Software Visualization* 1.212 (2005), pp. 67–75. DOI: 10.1145/1056018.1056027.

[68] G Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

[69] Maurice Termeer et al. „Visual exploration of combined architectural and metric information“. In: *Proceedings - VISSOFT 2005: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* 2005 (2005), pp. 21–26. DOI: 10.1109/VISSOF.2005.1684298.

[70] Richard Wettel and Michele Lanza. „Visualizing software systems as cities“. In: *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2007), pp. 92–99. DOI: 10.1109/VISSOF.2007.4290706.

[71] Alfred Inselberg and Bernard Dimsdale. *Parallel Coordinates*. 1991. DOI: 10.1007/978-1-4684-5883-1_9.

[72] Mark Seemann. *Dependency injection in. NET*. Manning New York, 2012.

[73] Gavin Bierman, Martín Abadi, and Mads Torgersen. „Understanding TypeScript“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8586 LNCS (2014), pp. 257–281. ISSN: 16113349. DOI: 10.1007/978-3-662-44202-9_11.

[74] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[75] Gaston C Hillar. *Building RESTful Python Web Services*. Packt Publishing Ltd, 2016.

[76] Glenn E Krasner, Stephen T Pope, et al. „A description of the model-view-controller user interface paradigm in the smalltalk-80 system“. In: *Journal of object oriented programming* 1.3 (1988), pp. 26–49.

[77] Chris Anderson and Anthony Gerard Champion. *Pro business applications with silverlight 4*. Springer, 2010.

[78] Michele Bertoli. *React Design Patterns and Best Practices*. Packt Publishing Ltd, 2017.

[79] Marc Garreau and Will Faurot. *Redux in Action*. Manning Publications Co., 2018.

[80] Steven John Simon. „The impact of culture and gender on web sites: an empirical study“. In: *ACM SIGMIS Database: the DATABASE for Advances in Information Systems* 32.1 (2000), pp. 18–37.

[81] John M. Carroll and Mary Beth Rosson. „Usability engineering“. In: *Computing Handbook, Third Edition: Information Systems and Information Technology* (2014), pp. 32–1–32–22. DOI: 10.1201/b16768.

[82] James V Bradley. „Complete Counterbalancing of Immediate Sequential Effects in a Latin Square Design Author (s): James V . Bradley Source : Journal of the American Statistical Association , Vol . 53 , No . 282 (Jun ., 1958), pp . 525- COMPLETE COUNTERBALANCING OF IMMED“. In: 53.282 (2014), pp. 525–528.

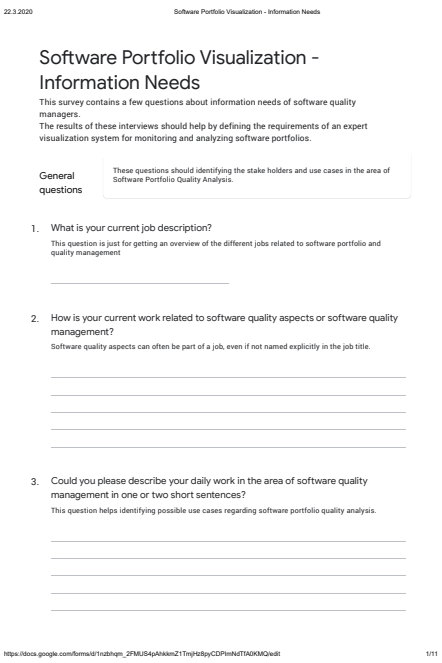
[83] Sam Kusic. „SUS - A quick and dirty usability scale“. In: *Iron and Steel Technology* 15.8 (2018), pp. 41–47. ISSN: 15470423. DOI: 10.5948/upo9781614440260.011.

[84] Aaron Bangor, Philip T. Kortum, and James T. Miller. „An empirical evaluation of the system usability scale“. In: *International Journal of Human-Computer Interaction* 24.6 (2008), pp. 574–594. ISSN: 10447318. DOI: 10.1080/10447310802205776.

[85] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. „Reporting experiments in software engineering“. In: *Guide to advanced empirical software engineering*. Springer, 2008, pp. 201–228.

Appendix

Questionnaire Software Portfolio Visualization - Information Needs



22.3.2020 Software Portfolio Visualization - Information Needs

Software Portfolio Visualization - Information Needs

This survey contains a few questions about information needs of software quality managers.
The results of these interviews should help by defining the requirements of an expert visualization system for monitoring and analyzing software portfolios.

General questions

These questions should identifying the stake holders and use cases in the area of Software Portfolio Quality Analysis.

1. What is your current job description?
   This question is just for getting an overview of the different jobs related to software portfolio and quality management
2. How is your current work related to software quality aspects or software quality management?
   Software quality aspects can often be part of a job, even if not named explicitly in the job title.
3. Could you please describe your daily work in the area of software quality management in one or two short sentences?
   This question helps identifying possible use cases regarding software portfolio quality analysis.

https://docs.google.com/forms/d/1nzbhqm_2FMUS4pAhkkmZ1TmjHz8pyCDPImNdTfA0KMQ/edit 1/11

Definition: Software Portfolio

A software portfolio describes a set of software projects/repositories that are grouped together in that way, that monitoring and comparing the quality attributes within this group would deliver meaningful results. Examples:

- The product suite developed by a company if monitored with company-wide quality standards

- Different, not directly related projects that are comparable on a technical or domain level

- A single software system, available in different, client-specific variations and releases (i.e. Framework based architectures etc...)

4. How many software repositories are (on average) part of your software portfolio?

This question should help to define the average scope of a software portfolio

_____

5. How large is the biggest project within your portfolio (in Lines of Code)?

This question should help to define the dimensions which have to be handled during software portfolio analysis.

Markieren Sie nur ein Oval.

- ◯ Less than 10 000 LOC
- ◯ 10 000 to 100 000 LOC
- ◯ 100 000 to 500 000 LOC
- ◯ 500 000 to 1 Mio LOC
- ◯ More than 1 Mio LOC

Software Portfolio Quality Tools

The goal here is to collect some information about the current market situation and popular features

6. Do you know any of the following Software Quality Managment Tools?

multiple answers possible

Wählen Sie alle zutreffenden Antworten aus.

- [ ] SonarQube (https://www.sonarqube.org/)
- [ ] SonarCloud (https://sonarcloud.io/)
- [ ] CodeClimate (https://codeclimate.com/)
- [ ] Codacy (https://www.codacy.com/)
- [ ] GrimoireLab (https://chaoss.github.io/grimoirelab/)
- [ ] VeraCode (https://www.veracode.com/)

7. Which tools do you use for your software quality tasks?

These can be all kind of tools, like specific dedicated software quality analysis applications like the ones in the question before, but also programs like Excel, Word or any general purpose or domain specific programming languages.

8. Regarding the tools you use, which features related to software quality analysis do you value most in these tools?

please give a short description of your favorite featue and why it is so valuable for you

9. What would be a feature you wish would be supported by your tool of choice but currently is not?

please give a short description of the feature you wish would be available in any tool

Visualization

Questions related to the visualization of software portfolio analysis data.

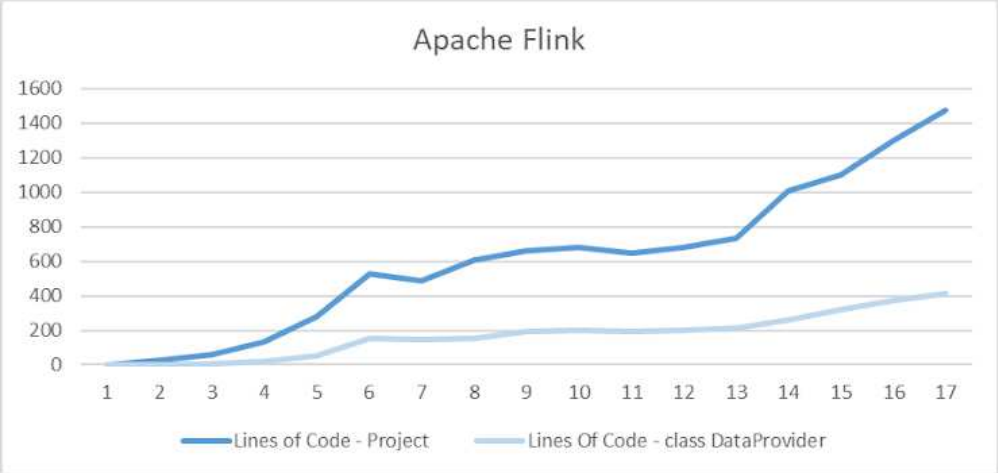
Feature Evaluation

Now comes a list of possible features a software portfolio analysis tool could provide.
The features will start at a very granular level (repositories and artifacts within repositories) and will then move up to a bird-eye Portfolio view.

For every feature, please rate on a scala from 1 (very important for you) to (5 = not important for you), how much you would like to have this feature available in your tool of choice.

10. Comparing metrics of single artefacts in relation to the project/portfolio

For example, to check how the metrics of a single file or class evolves compared to the overall trend of a project or portfolio

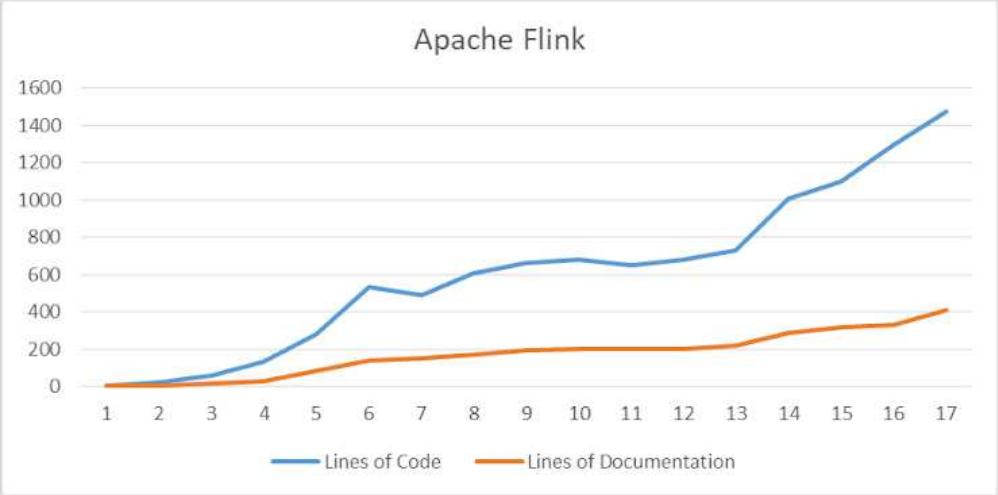


Markieren Sie nur ein Oval.

	1	2	3	4	5	
very important	◯	◯	◯	◯	◯	not important at all

11. Comparing metrics within the same project

For example to find correlations between different metrics (like Cyclomatic Complexiy and LOC) within a given project

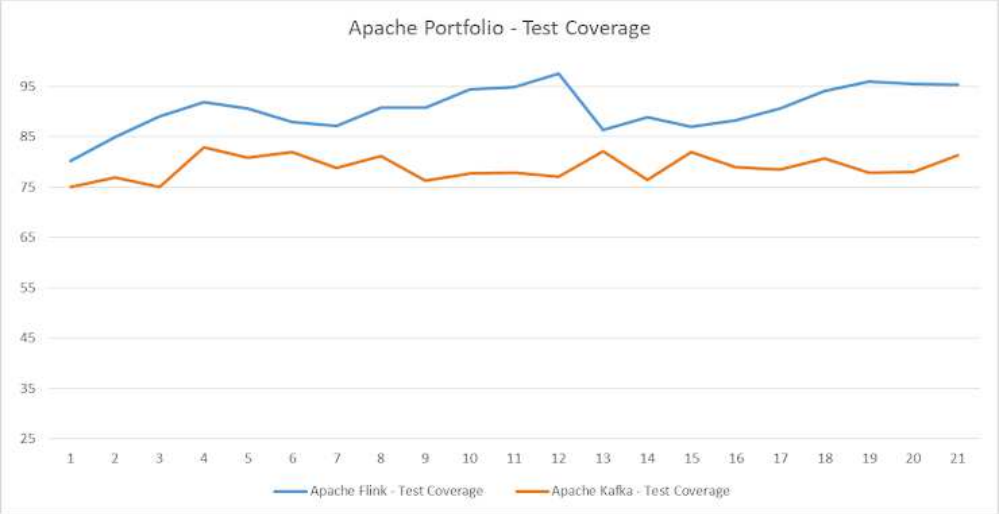


Markieren Sie nur ein Oval.

	1	2	3	4	5	
very important	◯	◯	◯	◯	◯	not important at all

12. Comparing metric trends of different projects within the same portfolio

For example to check how the metrics of one projects have evolved compared to other projects



Markieren Sie nur ein Oval.

	1	2	3	4	5	
very important	◯	◯	◯	◯	◯	not important at all

13. Determining the overall portfolio health by a custom aggregated indicator (like a star-rating or grading scheme)

Which metrics or aspects of a project or portfolio should be taken into account by such an indicator?

14. How valuable would you rate such an indicator to determine the overall portfolio/project health?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very important	◯	◯	◯	◯	◯	not important at all

15. Navigate to "Hot Spots" within the trend graph

Automatically navigate to points in time where the trend graph changed significantly (e.g. a metric went under a threshold, the change of a metric was more than 10% etc..., the relation between two metrics changed significantly)

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very important	◯	◯	◯	◯	◯	not important at all

16. What could be a possible Project or Portfolio "Hot Spots" that would be interesting for you?

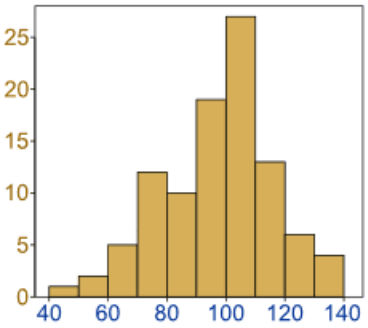
Metrics

This section is about the software metrics which are gathered and used to evaluate the health state of a software portfolio.

17. Which are the five most important metrics you use for your software quality analysis?

Please name or describe these metrics

18. Regarding cumulated metric values for a project or portfolio, in which values would you be interested in?



Distribution

Wählen Sie alle zutreffenden Antworten aus.

	Very interested	Interested	Neither	Not interested	Not interested at all
Average values (for instance, average Inheritance deep of all classes in a project)	☐	☐	☐	☐	☐
Median values (for instance, median of the inheritance deep for all classes in a project)	☐	☐	☐	☐	☐
Min/Max values (like the minimum/maximum test coverage a project can have over time)	☐	☐	☐	☐	☐
Distribution (for example, how are the lines of code per file distributed in a project)	☐	☐	☐	☐	☐
Thresholds (when does a specific metric violate a given threshold?)	☐	☐	☐	☐	☐

19. Do you have any specific thresholds for your metrics you use to define the health of a project? If yes, how have you determined the concrete values for your thresholds?

Please choose a possible answer for how you got your threshold values

Markieren Sie nur ein Oval.

- ◯ From personal experience
- ◯ From literature
- ◯ Suggested values from analysis tools like SonarQube
- ◯ I don't use any metric thresholds
- ◯ Sonstiges: _____

Questionnaire Software Portfolio Visualization - Scenario Based Expert Evaluation

Software Portfolio Visualization - Scenario Based Evaluation

This questionnaire will be used to evaluate the quality of an expert visualization system for analyzing software portfolio quality metrics.
This survey consists of four parts:

1. Demographical questions
2. Test Scenarios
3. Research Questions
4. System Usability Scale (SUS) questionnaire

Demographic Data and Experience

The visual perception of a person might be affected by many different characteristics.
As this prototype is based mainly on different visualization concepts, collecting additional demographic information might be helpful for the evaluation.
The collected data will be used only for evaluating the visualization prototype.
Answering these questions is optional.

1. What is your age?

_____

2. What is your gender?

Studies have shown that the visual perception might be affected by a person's gender. As this prototype is based mainly on different visualization concepts, collecting this information might be helpful for the evaluation.

Markieren Sie nur ein Oval.

- female
- male
- other

3. How would you quantify your experience in (general) software development in years?

Wählen Sie alle zutreffenden Antworten aus.

- [ ] < 1 Year
- [ ] 1 - 2 Years
- [ ] 2 - 5 Years
- [ ] 6 - 10 Years
- [ ] 11 - 15 Years
- [ ] 16 - 20 Years
- [ ] more than 20 Years

4. How would you quantify your experience in quality related software tasks in years?

Quality related tasks could be anything like collecting and analyzing software quality metrics, working with tools like SonarQube etc...). This could also include tasks like writing plugins or scripts for your quality analysis workflow.

Wählen Sie alle zutreffenden Antworten aus.

- [ ] < 1 Year
- [ ] 1 - 2 Years
- [ ] 2 - 5 Years
- [ ] 6 - 10 Years
- [ ] 11 - 15 Years
- [ ] 16 - 20 Years
- [ ] more than 20 Years

5. How much time of your average work day do you spend with quality related tasks (in percent)?

Quality related tasks are all tasks that are more focused on your quality management workflow like measuring or analyzing software metrics or configuring quality tools etc. Implementation tasks that are more feature related (like, for instance, writing unit tests for a feature) should not be considered here.

Scenarios

The following scenarios describe some possible real world scenarios you should try to solve by using the expert visualization.

Please take as much time as you need to complete the individual scenarios and do not hesitate to ask if something seems unclear.

Please also note that the only goal of this questionnaire is to evaluate the functionality and quality of the prototype and not your competence level, so there are no correct or wrong answers, as any answer might help us in evaluating the prototype.

Scenario 1 / 7: Regarding Maven, do the changes of the average Lines of Code metric correlate stronger with the changes of the average Comment to Code Ratio or with the average Cyclomatic Complexity metric?

When comparing metrics with a different value range, it can often be better to compare their relative changes instead to find possible relations between the trends.

Remark: Please note that the "Source Lines of Code" metrics automatically sets its aggregate value to "sum" when selected, so every time you select the "Source Line of Code" metric, you have to manually reset the aggregate value to "Average".

Please answer the following two questions BEFORE starting the task

6. How would you rate the difficulty of this task when using only existing tools that you know (like GitHub, SonarQube, Excel etc...)

This question helps us to compare the efficiency of the prototype with existing solutions

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very difficult to solve	◯	◯	◯	◯	◯	very easy to solve

7. How much time would you approx. need to solve this task with your exsting tools?

Markieren Sie nur ein Oval.

- ◯ Less than 5 minutes
- ◯ 5 - 10 minutes
- ◯ 10 - 30 minutes
- ◯ 30 minutes - 1 hour
- ◯ more than one hour
- ◯ Sonstiges: _____

Please answer these questions AFTER solving this task

8. How relevant would you rate this scenario?

Would you say this scenario represents a real possible use case or would you say it is rather not relevant for your work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
totally irrelevant for me	◯	◯	◯	◯	◯	very relevant for me

Scenario 2 / 7:
Regarding their latest trend, which projects have a median lack of cohesion (LCOM) that is higher than 65?

The Single Responsible Principle suggests that classes should at best only have one single purpose. The Lack of Cohesion metric can be used to measure this feature. During this scenario, it should be tested how the tool could support users by setting quality constraints on metric trends and monitoring them.

Please answer the following two questions BEFORE starting the task

9. How would you rate the difficulty of this task when using only existing tools that you know (like GitHub, SonarQube, Excel etc...)

This question helps us to compare the efficiency of the prototype with existing solutions

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very difficult to solve	◯	◯	◯	◯	◯	very ease to solve

10. How much time would you approx. need to solve this task with your exsting tools?

Markieren Sie nur ein Oval.

- ◯ Less than 5 minutes
- ◯ 5 - 10 minutes
- ◯ 10 - 30 minutes
- ◯ 30 minutes - 1 hour
- ◯ more than one hour
- ◯ Sonstiges: _____

Please answer these questions AFTER solving this task

11. How relevant would you rate this scenario?

Would you say this scenario represents a real possible use case or would you say it is rather not relevant for your work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
totally irrelevant for me	◯	◯	◯	◯	◯	very relevant for me

Scenario 3 / 7: Regarding its latest trend, which project has an artefact (file or class) that has a maximum cyclomatic complexity of more than 50?

Metric thresholds can not only be applied on project level, but also on a more fine grained artifact level, like on files or classes. This scenario evaluates the possibility of the tool to monitor metrics on a very fine detail level by still keeping an overview of the whole portfolio.

Please answer the following two questions BEFORE starting the task

12. How would you rate the difficulty of this task when using only existing tools that you know (like GitHub, SonarQube, Excel etc...)

This question helps us to compare the efficiency of the prototype with existing solutions

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very difficult to solve	◯	◯	◯	◯	◯	very easy to solve

13. How much time would you approx. need to solve this task with your exsting tools?

Markieren Sie nur ein Oval.

- ◯ Less than 5 minutes
- ◯ 5 - 10 minutes
- ◯ 10 - 30 minutes
- ◯ 30 minutes - 1 hour
- ◯ more than one hour
- ◯ Sonstiges: _____

Please answer these questions AFTER solving this task

14. How relevant would you rate this scenario?

Would you say this scenario represents a real possible use case or would you say it is rather not relevant for your work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
totally irrelevant for me	◯	◯	◯	◯	◯	very relevant for me

Scenario 4 / 7: Keep the portfolio state to green even if RocketMQ does violate the previously defined cyclomatic complexity rule.

Portfolios can contain different projects with various complexities and requirements, it should therefore be possible to evaluate their health state individually if necessary. There could also exist prototypes or project in early development, which, while still be monitored, should not affect the overall health of the portfolio.

Please answer the following two questions BEFORE starting the task

15. How would you rate the difficulty of this task "setting individual metric thresholds for individual projects" when using only existing tools that you know (like GitHub, SonarQube, Excel etc...)

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very difficult to solve	◯	◯	◯	◯	◯	very easy to solve

16. How much time would you approx. need to solve this task with your exsting tools?

Markieren Sie nur ein Oval.

- ◯ Less than 5 minutes
- ◯ 5 - 10 minutes
- ◯ 10 - 30 minutes
- ◯ 30 minutes - 1 hour
- ◯ more than one hour
- ◯ Sonstiges: _____

Please answer these questions AFTER solving this task

17. How relevant would you rate this scenario?

Would you say this scenario represents a real possible use case or would you say it is rather not relevant for your work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
totally irrelevant for me	◯	◯	◯	◯	◯	very relevant for me

Scenario 5 / 7: Regarding Maven and RocketMQ, which is the highest class coupling value a class has and which class is it?

If a class has too many couplings, it can be difficult to maintain the class as all the dependencies to other classes have to be kept in mind. A class with high couplings could therefore be a possible subject for a refactoring. This scenario checks the possibilities of the tool provide navigation from a bird's eye portfolio view into the details of a single class.

Please answer the following two questions BEFORE starting the task

18. How would you rate the difficulty of this task when using only existing tools that you know (like GitHub, SonarQube, Excel etc...)

This question helps us to compare the efficiency of the prototype with existing solutions

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very difficult to solve	◯	◯	◯	◯	◯	very easy to solve

19. How much time would you approx. need to solve this task with your exsting tools?

Markieren Sie nur ein Oval.

- ◯ Less than 5 minutes
- ◯ 5 - 10 minutes
- ◯ 10 - 30 minutes
- ◯ 30 minutes - 1 hour
- ◯ more than one hour
- ◯ Sonstiges: _____

Please answer these questions AFTER solving this task

20. How relevant would you rate this scenario?

Would you say this scenario represents a real possible use case or would you say it is rather not relevant for your work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
totally irrelevant for me	◯	◯	◯	◯	◯	very relevant for me

Scenario 6 / 7: What might be a possible reason that the total lines of code of log4j and log4net became more equal?

Monitoring and comparing the evolution of two projects can be useful when making project related decisions. This scenario therefore evaluates the possibility to combine cross project trend analysis on
different levels.

Please answer the following two questions BEFORE starting the task

21. How would you rate the difficulty of this task when using only existing tools that you know (like GitHub, SonarQube, Excel etc...)

This question helps us to compare the efficiency of the prototype with existing solutions

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very difficult to solve	◯	◯	◯	◯	◯	very easy to solve

22. How much time would you approx. need to solve this task with your exsting tools?

Markieren Sie nur ein Oval.

- ◯ Less than 5 minutes
- ◯ 5 - 10 minutes
- ◯ 10 - 30 minutes
- ◯ 30 minutes - 1 hour
- ◯ more than one hour
- ◯ Sonstiges: _____

Please answer these questions AFTER solving this task

23. How relevant would you rate this scenario?

Would you say this scenario represents a real possible use case or would you say it is rather not relevant for your work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
totally irrelevant for me	◯	◯	◯	◯	◯	very relevant for me

Scenario 7 / 7: Regarding the Maven project, how does the amount of classes with one base class and classes with two base classes change between May 15, 2009 and Jun 20, 2020?

Monitoring the trend of metrics over time within a single project can provide meaningful insights about the evolution of a project. The number of base classes of a class could be an indicator for the reusability of common functionality (together with other metrics).

Remark: You can use the "Depth of Inheritance Tree" metric to determine the number of base classes a class has.

Please answer the following two questions BEFORE starting the task

24. How would you rate the difficulty of this task when using only existing tools that you know (like GitHub, SonarQube, Excel etc...)

This question helps us to compare the efficiency of the prototype with existing solutions

Markieren Sie nur ein Oval.

	1	2	3	4	5	
very difficult to solve	◯	◯	◯	◯	◯	very easy to solve

25. How much time would you approx. need to solve this task with your exsting tools?

Markieren Sie nur ein Oval.

- ◯ Less than 5 minutes
- ◯ 5 - 10 minutes
- ◯ 10 - 30 minutes
- ◯ 30 minutes - 1 hour
- ◯ more than one hour
- ◯ Sonstiges: _____

Please answer these questions AFTER solving this task

26. How relevant would you rate this scenario?

Would you say this scenario represents a real possible use case or would you say it is rather not relevant for your work?

Markieren Sie nur ein Oval.

	1	2	3	4	5	
totally irrelevant for me	◯	◯	◯	◯	◯	very relevant for me

Research Questions

The following section aims at answering some of the research questions which are to be examined in the context of this work.

27.

Markieren Sie nur ein Oval pro Zeile.

	1 (Strongly disagree)	2	3	4	5 (Strongly agree)
The interface was well suited for solving the tasks	◯	◯	◯	◯	◯
The amount of information was always manageable	◯	◯	◯	◯	◯

System Usability Scale (SUS)

The System Usability Scale (SUS) is a simple, ten-item scale giving a global view of subjective assessments of usability (Brooke, John. "SUS: a “quick and dirty'usability." Usability evaluation in industry (1996): 189)

28.

Markieren Sie nur ein Oval pro Zeile.

	1 (Strongly disagree)	2	3	4	5 (Strongly agree)
I think that I would like to use this system frequently					
I found the system unnecessarily complex					
I thought the system was easy to use					
I think that I would need the support of a technical person to be able to use this system					
I found the various functions in this system were well integrated					
I thought there was too much inconsistency in this system					
I would imagine that most people would learn to use this system very quickly					
I found the system very cumbersome to use					
I felt very confident using the system					
I needed to learn a lot of things before I could get going with this system					

Diagram control options

min/max: show min max values of recorded metrics

scale: use same y-axis range for all projects

thresholds: show metric thresholds (if configured through health indicator)

switch between absolute metric values and relative changes since beginning

switch the aggregate function that is visualized:

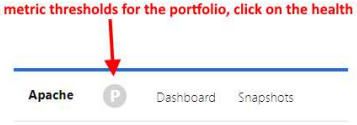
sum: cummultative metric values
works only for Source Lines of Code (SLOC)

avg/med: average or median of all recorded metric values for the given snapshot

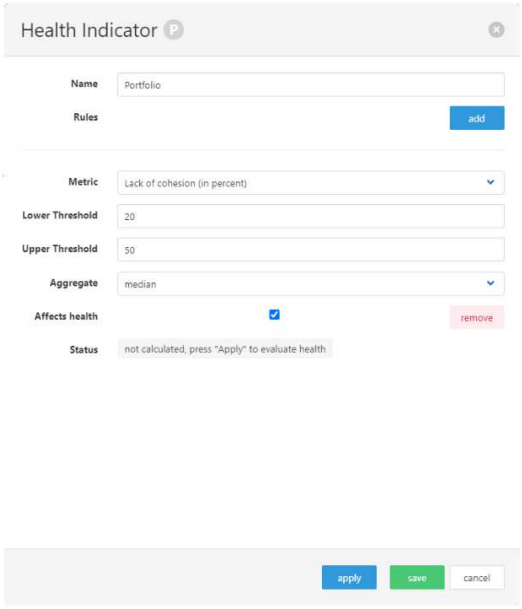
Remark:
When choosing SLOC and another metric in the same diagram, SLOC will by default always be shown as "sum", the other metric as "median", to change both to "avg" or "med", you have to click the corresponding button again

Defining metric thresholds

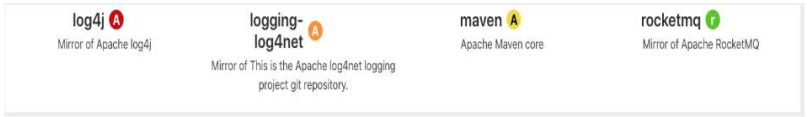
1. To define metric thresholds for the portfolio, click on the health indicator symbol



2. Define new metric rules that will be applied to all proejcts in the portfolio



3. Press "apply" or "save" to activate the newly defined rule.
The indicator symbol should reflect whether the metric rule is violated



Critical: The recent metric trends are violating the thresholds

Worsening: The threshold is not yet violated, but the trend is getting worse

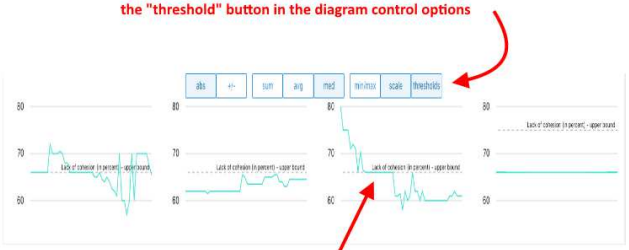
Improving: The threshold is not violated and the trend is also moving away

Good: The trend values are in the optimal area

4. To customize individual rules for a specific project, click on the health indicator next to the project's name and choose "Create a new indicator for this project"

After defining a custom project indicator, the indicator symbol's letter should change in the project list (see the small "r" in constract to the "A" in step 3.

5. To visualize the metric thresholds in the diagram, press the "threshold" button in the diagram control options



a dashed line should now indicate the metric threshold