

Katherina Nizenkov

Design and implementation of a developer-centric quality web application

Bachelor's thesis

Supervisors:

Prof. Dr. Barbara Paech

Anja Kleebaum

Institute of Computer Science

University of Heidelberg

Johannes Häußler

SAP SE

June 30, 2019

Zusammenfassung

[Motivation] Bei SAP HANA gibt es zahlreiche Tools zur Qualitätssicherung des Softwareentwicklungsprozesses. Oftmals ist die Sicht solcher Tools eher auf die Bedürfnisse der Produktmanager und Qualitätsmanagementbeauftragten zugeschnitten. Die Ergebnisse sind oft zusammengefasst und nicht aktuell, weil die Datenerhebung der Qualitätsmetriken vom Release-Zeitpunkt abhängig ist. Diese Sicht auf die Qualitätssicherung erfüllt aber nicht immer die Informationsbedürfnisse eines Softwareentwicklers bzw. einer Softwareentwicklerin.

[Ziele] Das Hauptziel dieser Arbeit ist der Entwurf und die Implementierung einer Webanwendung, die die Qualitätsmetriken anzeigt, die aus Entwicklersicht nützlich sind. Diese Metriken sollen sich direkt in konkrete Aufgaben übersetzen lassen. Um dieses Ziel umzusetzen, wurde eine Literaturrecherche zum Thema entwickler-zentrierte Qualitätssicherung durchgeführt und anschließend wurden die potenziellen NutzerInnen einer solchen Webanwendung in Einzelinterviews zu den Metriken befragt.

[Ergebnisse] Die Hauptergebnisse dieser Arbeit sind zum einen eine Liste von Metriken, die sich bei den Interviews als nützlich herausgestellt haben und zum anderen ein Prototyp der Webanwendung (*Quality Page*), der zwei dieser Metriken anzeigt. Die implementierte Quality Page wurde von den Teilnehmern der Interviewstudie bewertet und im Allgemeinen für nützlich befunden. Während der Evaluation wurden zudem Ideen genannt, wie die Quality Page weiter ausgebaut werden kann, um EntwicklerInnen täglich bei der Qualitätssicherung ihres Codes unterstützen zu können.

Abstract

[Motivation] Numerous tools that support quality assurance (QA) of the software development process exist at SAP HANA. Often, these tools reflect the perspective of the product managers and QA engineers. The analysis results are often summarized and are not up-to-date as the data collection of quality metrics depends on the release schedule. This delivery- and QA-centric view on software quality does not always meet the information needs of software developers.

[Goals] The overall goal of this work is design and implementation of a web application to display quality metrics which developers consider useful for their daily work. These metrics should be directly transferable into concrete tasks. To achieve this goal, we conducted a literature study on developer-centric quality assurance. Subsequently, we carried out one-on-one interviews about quality metrics with the potential users of the web application.

[Results] One of the two main results of this work is a list of metrics that have proven themselves useful based on the results of the interviews. The second main result is a prototype of the web application (*Quality Page*) which displays two of these metrics. The implemented Quality Page was evaluated by the participants of the interview study and, for the most part, was considered useful. Moreover, during this evaluation, the participants exchanged ideas on how to extend the Quality Page so that it would support developers in their quality assurance efforts on a daily basis.

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Goals of the Project	5
1.2.1	Goal 1: Problem Investigation	5
1.2.2	Goal 2: Treatment Design	5
1.2.3	Goal 3: Treatment Validation	5
1.3	Outline	6
2	Background	7
2.1	Basic Concepts	7
2.1.1	Software Quality Assurance	7
2.1.2	Software Quality Metrics	8
2.2	Current state at SAP HANA	11
2.2.1	Process of Quality Assurance at SAP HANA	12
2.2.2	Existing Code Quality Metrics and Tool Support at SAP HANA	12
3	Literature Study	14
3.1	Research Questions	14
3.2	Research Strategy	15
3.3	Research Results	16
3.4	Comparison of Q-Rapids, Squale, Quamoco and Tricorder	18
3.4.1	Used Standard	22
3.4.2	Targeted Roles	22
3.4.3	Approach Description	22
3.4.4	Used Code-Related Metrics (RQ1)	23
3.4.5	Actionability of Metrics (RQ2)	23
3.4.6	Visualization of Metrics (RQ3)	24
3.4.7	Evaluation	25
3.5	Answers to the Research Questions	27
3.5.1	Metrics Software Developers Need (RQ1)	27
3.5.2	Transferring Metric into a Task (RQ2)	28

3.5.3	Visualization of a Metric (RQ3)	28
3.6	Insights for This Work	28
3.6.1	Stakeholder Interviews and Feedback Mechanism to Determine Useful Metrics	29
3.6.2	Actionable through Thresholds and Drill-Down to Code	29
3.6.3	Visualization and Customization	30
4	Interview Study	31
4.1	Method	31
4.1.1	Study Design	31
4.1.2	Planning and Preparation of Interviews	31
4.1.3	Data Collection	34
4.2	Results and Discussion	34
4.2.1	Answers to the Interview Questions	35
4.2.2	Answers to the Research Questions	39
4.2.3	Insights for This Work	40
5	Requirements	42
5.1	Persona	42
5.2	Functional Requirements	43
5.2.1	Tasks of the User	43
5.2.2	System Functions	44
5.2.3	Domain Data	48
5.3	Non-Functional Requirements	49
5.4	User Interface Structure Diagram	49
5.5	Summary	50
6	Design and Implementation	52
6.1	Architecture of the Web Application	52
6.1.1	Landing Page vs. IDE Plug-In	52
6.1.2	Frontend and Backend Technologies	53
6.2	Class Diagram	54
6.3	Functional Requirements	55
6.3.1	SF Show current metric value	56
6.3.2	SF Filter files	57
6.3.3	SF Sort files	58
6.3.4	SF Search for files	59
6.3.5	SF Hide / unhide a metric	60
6.3.6	SF Show trend view	62
6.3.7	SF Create and update threshold for a metric	63

6.3.8	SF Save selected filters	63
6.3.9	SF Apply saved filters	64
6.4	Non-Functional Requirements	64
6.4.1	Performance	64
6.4.2	Reusability	65
7	Quality Assurance	66
7.1	Continuous Integration and Tests	66
7.2	Static Tests	67
7.3	Dynamic Tests	67
7.3.1	Unit Tests	67
7.3.2	System Tests with Mock Server	68
7.3.3	NFR: Performance efficiency (Time-behavior)	72
8	Evaluation	73
8.1	Feedback Meeting	73
8.1.1	Method	73
8.1.2	Results	73
8.2	Discussion and Future Work	75
9	Conclusion	76
9.1	Summary and Discussion	76
9.2	Outlook	77
10	Bibliography	78
	List of figures	81
	List of tables	83
	Glossary	84
	Acronyms	85

1 Introduction

Industrial software development today is performed under enormous time pressure [9]. Nevertheless, high code quality is expected despite the tight release schedules. There exists a multitude of tools which can make the job of a software developer easier, such as auto-completion in the IDE or suggestion of quick fixes for warnings. Nevertheless, developers still have to answer a lot of questions and make a lot of decisions regarding code quality by themselves. This thesis investigates on how to provide developers with specific metrics such as e.g. Cyclomatic complexity or number of Lines of Code (LOC) to support them during this decision-making process and in improving code quality.

1.1 Motivation

This work was written in cooperation with SAP SE in Walldorf, where hundreds of software engineers work on highly complex functionalities of SAP HANA, an in-memory database management system. The more software developers work together, the more sophisticated and intensive quality management efforts have to be. Currently, there is an extensive system in place at SAP HANA that provides information relevant for ensuring high product quality. Originally, this system was designed primarily for product managers and quality assurance engineers. However, individual software developers and small teams of software developers are often not satisfied with the granularity of such reporting. They perceive overviews and summaries, which may be very helpful for product managers and quality assurance engineers, as too coarse-grained or too abstract. In particular, a statement like “number of defects has to be reduced by 10 % in the next two months” is not actionable, i.e. a developer cannot directly translate it into a concrete task. Moreover, developers have different needs with respect to the timeliness of quality reporting: Whereas managers are more interested in the post-release statistics, developers prefer to receive quality data before a release and in general, more often than managers, ideally even before their new code is merged into the database.

1.2 Goals of the Project

With this work, we attempt to find a developer-centric approach to software quality metrics. Developers should be able to directly transform any of the metric values into concrete development tasks. To this purpose, a tool prototype should be designed. This prototype should lay the foundation for a new quality application providing transparency on the intrinsic code quality aspects such as readability and maintainability. This application should motivate software developers to continuously improve their code. To ensure this, the application should enable them to see the impact of such improvements and to plan them in fast iterations such as a two-week SCRUM sprint, for example.

With this general objective in mind, we derive three goals for this project, following the approach for structuring a design science project as described by Wieringa in *Design Science Methodology for Information Systems and Software Engineering* [22]. Wieringa decomposes a design task into three phases: problem investigation, treatment design, and treatment validation. This set of three tasks is called the design cycle, as researchers iterate over these tasks many times during a design science research project.

1.2.1 Goal 1: Problem Investigation

During problem investigation, the aim is to learn about stakeholder goals and to understand the problem to be treated. For the investigation of our research problem, we combine two methods: a study of scientific literature and an interview-based study with the stakeholders.

1.2.2 Goal 2: Treatment Design

Based on the knowledge gained during the problem investigation task, we define functional and non-functional requirements for a tool prototype of a web application, which we from hereon refer to as *Quality Page*. We then implement these requirements.

1.2.3 Goal 3: Treatment Validation

The designed Quality Page is presented to the interviewed stakeholders to confirm that this tool would be useful for them. The treatment validation completes the design cycle.

The results of the validation can be a starting point of the next design cycle and basis for future work on the Quality Page.

1.3 Outline

This thesis is structured as follows: Chapter 2 introduces the basic concepts needed to understand the context of this work and gives an overview on the quality assurance process at SAP HANA. Chapter 3 describes the study of relevant literature. In Chapter 4, the interview-based case study is presented. The requirements derived from both studies are documented in Chapter 5. Chapter 6 is dedicated to the design and implementation of the Quality Page. In Chapter 7, the performed quality assurance measures for the development process of the application are listed. Chapter 8 presents the carried out evaluation and Chapter 9 concludes the thesis.

2 Background

With this chapter, we put software metrics into the general context of software quality assurance and provide other background information needed to understand the rest of this thesis, as e.g. the description of the quality assurance process at SAP HANA.

2.1 Basic Concepts

In this section, we introduce the concepts of software quality assurance and software quality metrics.

2.1.1 Software Quality Assurance

According to the *Guide to the Software Engineering Body of Knowledge*, software quality assurance is “a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate and produce software products of suitable quality for their intended purposes” [3].

But what is “suitable quality” of a software product? The ISO/IEC 9126 standard and the subsequent ISO/IEC 25010 standard try to capture what software quality is by dividing it into multiple product quality characteristics. More recent ISO/IEC 25010 names following eight: *functional suitability*, *reliability*, *usability*, *security*, *compatibility*, *portability*, *maintainability* and *performance efficiency*.

Each of these characteristics can again be decomposed into other sub-characteristics. E.g. *maintainability* can be seen as a combination of the quality aspects *modularity*, *reusability*, *analysability*, *modifiability* and *testability*.

To be able to evaluate these quality characteristics, they first have to be made measurable. To this purpose, software quality metrics can be used.

2.1.2 Software Quality Metrics

The authors of *IEEE Standard for a Software Quality Metrics Methodology* define software quality metric as a “function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality” [6].

From now onward, we will use the term *metric* synonymously with *software quality metric*.

As the focus of this work lies on making metrics actionable for developers, we want to concentrate on those metrics which developers can influence directly. A developer cannot change the number of future bugs or number of future system crashes (at least not directly), she or he can change number of LOC in a file by dividing it in two files, for example. Before we can explore the aspects of usefulness and actionability of metrics for developers, we first have to acquire a list of potentially relevant metrics that we then can examine during our literature and interview study.

In his book *Metrics and models in software quality engineering* [7], Kan points out that such metrics as number of defects or average duration of time needed to remove a defect do not consider the “internal dynamics of design and code of the software”. Kan defines a group of metrics which he calls “complexity metrics” or “metrics about design and code implementation”. In his opinion, these metrics are relevant for the software engineers as they offer more clues on how to improve the quality of their code as opposed to such metrics that are solely based on the external behavior of the program like number of reported defects or number of system failures in the last week. The latter tend to treat the software as a black box and do not offer an internal view on software.

We decided to include following of Kans complexity metrics in our preliminary list, which we then will use for our interview study:

1. Number of LOC (for a file and for a method)
2. Complexity metrics: Halstead’s Software Science and Cyclomatic Complexity
3. Object-oriented metrics based on the metrics suite proposed by Chidamber and Kemerer [4]

Additionally, we analyzed metrics provided by an open-source program analysis platform SonarQube¹, which is widely used in industrial practice and was also utilized to collect

¹<https://sonarcloud.io/documentation/user-guide/metric-definitions> (accessed June 26, 2019)

raw data for the Q-Rapids project [11] encountered in our literature study described in Chapter 3.

We complemented our list with following metrics offered by SonarQube, which were also used in the Q-Rapids project [11]:

1. Comment percentage
2. Duplicate percentage
3. Number of code style violations

Additionally, we decided to include the following metrics related to the version control history, as they get increasingly more attention [17]:

1. Frequently changed files (Q-Rapids [11])
2. Files often changed together (logical dependencies [1])
3. Number of developers who changed the file in the last month/year (code ownership [14])

Test coverage is also one of the metrics assessed in the Q-Rapids project, which we also adopted for our list. In the context of this work, we will consider test coverage percentage for a file and we will differentiate between three different coverage types: unit, full and diff. Coverage achieved through unit tests will be hereon referred to as *unit coverage* and the coverage resulting from unit tests, integration tests and end-to-end tests as *full coverage*. Both unit and full coverage values can be calculated for either entire code or just for the newly submitted code. The latter will be referred to as *diff coverage*. In the context of this project, if not explicitly specified, by coverage we always imply line coverage - as opposed to e.g. branch coverage (coverage for each executed branch in the control structure of the program).

Table 2.1 provides the entire overview of metrics that we will discuss regarding their actionability and usefulness for software developers. This list will be used in the interview study described in Chapter 4.

Table 2.1: Overview of code-relevant metrics.

Metric	Description
Number of LOC (file)	Number of executable lines of code in a file. Represents the size and complexity of a program.

Number of LOC (method)	Number of executable lines of code in a method. Represents the size and complexity of a method.
Comment percentage	Number of comment lines divided by the total number of lines in a file. High number indicates good readability and understandability.
Cyclomatic complexity	Number of linearly independent paths in a program. Low number indicates testability and understandability of a program.
Halstead metrics	System of equations based on number of operands or operators in a program. Represents software size and complexity, even estimates development effort and number of future defects.
Frequently changed files	Number of times a file was changed in a certain time period. High number indicates problematic code parts (“hotspots”).
Files changed together	Listed files that are always changed when the given file is changed. Indicates logical dependencies between files.
Number of authors	Number of developers who changed the given file in a certain period of time. If high, indicates missing decentralization.

Object-oriented metrics	<p>Number of methods in a class: High number indicates a “God” class.</p> <p>Depth of Inheritance Tree (DIT): length of the maximum path of a class hierarchy from the node to the root of the inheritance tree. High number indicates low testability.</p> <p>Number of Children of a Class (NOC): number of immediate successors (subclasses) of a class in the hierarchy. High number indicates low testability.</p> <p>Coupling Between Object Classes (CBO): number of classes to which a given class is coupled. An object class is coupled to another one if it invokes another one’s member functions or instance variables. High number indicates high complexity.</p> <p>Response for a Class (RFC): number of methods that can be executed in response to a message received by an object of that class. High number indicates complexity of a class.</p> <p>Lack of Cohesion on Methods (LCOM): number of disjoint sets of local methods. Low cohesion indicates complexity and error-proneness.</p>
Duplicate percentage	Duplicated lines divided by the total number of lines of code per file. High number indicates low changeability and maintainability.
Number of code style violations	Number of code style rule violations per file. High number indicates error-proneness.
Coverage percentage	Number of LOC covered by tests divided by the total number of LOC per file. High coverage percentage is associated with less future errors.

2.2 Current state at SAP HANA

In this section, we describe the current situation around quality assurance process and visualization of metrics at SAP HANA. This helps us prepare for the interview study, in which we ask the interviewees about how satisfied they are with the existing tools.

2.2.1 Process of Quality Assurance at SAP HANA

In the following, we provide a simplified description of the process of quality assurance at SAP HANA. The process is more complex than illustrated here and may vary depending on the development team. Therefore, we focus only on the aspects relevant to understand the context of this work.

First, an individual developer submits his or her new code to the version control system (VCS). This new code is called a change and has a unique identifier. Subsequently, an automated server runs tests to ensure that the added code will not break existing functionality. Additionally, a program analysis tool automatically checks the change and if problems are detected, the developer receives warnings in form of an email notification. At this point, the developer has the possibility to change the code to address the warnings resulted from both the test run and the analysis tool. Afterwards, the developer can submit the code again. After completing this step, the developer requests members of his or her team to review the code. The new code can be merged into the code base only after the approval of the appointed reviewers. After the merge, the developer can view the metrics for the code in the existing central tools, which are described in the following section 2.2.2.

2.2.2 Existing Code Quality Metrics and Tool Support at SAP HANA

This section describes existing visualization tools for code quality metrics at SAP HANA. Currently, the following web applications (landing pages) display information on code quality: the component-related platform, the coverage-related platform and an online code browser.

Component-related platform

The component-related platform provides metrics for each HANA component. These metrics include:

- Number of critical defects
- Line coverage (full, unit, diff full and diff unit as defined in Section 2.1.2)
- Issues detected by the static code analysis tool

- Compiler warnings
- Number of branches
- Library dependencies
- Overview of all tests and their respective run time

From the metric value on line coverage, the user can go directly to the Coverage-related platform for more details. Additionally, metrics on file level are provided for each component. The displayed metrics for each file include:

- Absolute number of executable LOC covered by tests
- Absolute number of executable LOC not covered by tests
- Line coverage percentage
- Issues detected by the static code analysis tool
- Compiler warnings

Coverage-related platform

The Coverage-related platform can either be reached from the Component-related platform by clicking on a coverage value for a HANA component or it can be accessed directly in the browser.

Additionally to the possibility to view the coverage value at a specific point in time, there is a trend chart showing the development of the coverage values and number of “relevant”² lines of code over a period of 15 months.

Online code browser

The page enables the user to browse the entire HANA repository for a desired source file and also offers the possibility to view the information on which revision and author last modified each line of a file.

²“Relevant” lines of code refer to the lines of code in files that do not belong to third party software and are not files containing tests.

3 Literature Study

To decide upon which metrics should be displayed in our web application and how they should be visualized, we conduct two studies: a study of relevant literature and an empirical study in which we interview potential users of the application. This chapter describes the systematic literature review undertaken according to the guidelines proposed by Kitchenham and Charters [8]. In Section 3.1, we specify research questions, which this study is supposed to answer. Then we define the research strategy in Section 3.2 and document the results of the search process in Section 3.3. The found articles are then compared in Section 3.4 and the answers to the research questions are described in Section 3.5. In conclusion, Section 3.6 presents insights gained from this study with respect to the requirements of our application.

3.1 Research Questions

The main goal of our metric-based quality application is to motivate software developers to continuously improve their code. This should be achieved through providing relevant metrics to identify potential areas for improvement. But simply knowing where the areas for improvement are is often not enough. Therefore, provided insights should be easily transferable into specific actions, or short - actionable. The impact of these actions should be visible to be truly motivating. Therefore, with our application the user should not only be able to plan an improvement but also to track the impact of this improvement in the context of an industrial code base.

From this goal, we derive three research questions (RQ) presented in Table 3.1.

Table 3.1: Research questions derived from the project’s goal.

RQ1	Which metrics do software developers need?
RQ2	How can metrics be transferred into concrete tasks?
RQ3	How can metrics be visualized?

RQ1 aims at identifying code quality metrics which in developer’s point of view are best

suited to provide information on code quality.

RQ2 focuses on the aspect of actionability: How can the displayed metric lay foundation for a concrete development task?

RQ3 focuses on visualization techniques and addresses the question of how to display code quality metrics in the context of a large code base. With SAP HANA's code base accounting for over 3.5 million lines of code, it is not exactly motivating to see that the impact of improvement efforts of your team - no matter how work-intensive they might have been - remains insignificant. Therefore, visualization mechanisms should be found to help developers not only identify areas for improvement in their code but also to display made progress in such a way that it motivates them.

The main objective of the literature study was to identify approaches in the existing literature which address these research questions.

3.2 Research Strategy

Through the initial research to gain basic knowledge in the area of software quality measurement, the article by Martínez-Fernández et al. [11] was found. The article not only presents a quality model which the authors developed in close cooperation with software practitioners but also describes the operationalization of the said model. Because of this combination of theoretical aspects of software quality and practical implementation of a quality model depicted in this article, it seemed to be a good starting point to search for other relevant literature sources.

A snowballing approach was applied to [11]. To this purpose, all articles referencing [11] and referenced by [11] were examined if they should be included in the list of relevant literature. Then the same procedure was conducted with the found articles. To limit the search to a manageable number of sources, the relevance criteria presented in Table 3.2 were formulated and applied during each iteration of the snowballing method.

The criteria 1 - 4 are standard criteria applied to literature selection at the Chair of Software Engineering [15]. The fifth criterion was formulated to select literature sources with content relevant for this specific work. The sixth criterion was introduced to focus on recent work due to the fast changing nature of research in the area of software engineering and actionable analytics in particular.

Table 3.2: Criteria for selection of relevant articles.

No.	Description
1	The article was peer reviewed
2	The article is available in English or German
3	The article is freely available in the existing databases or university library
4	The article has a clear focus on software engineering
5	The article contains information relevant for answering the RQs
6	The article was published in the last ten years

3.3 Research Results

In Table 3.3 the results of the snowballing iterations are documented. For instance, the article by Sadowski et al.[20] cites 43 articles (backward direction of snowballing), none of which fulfill all relevance criteria. The article is at the same time cited by other 28 articles (forward direction of snowballing), one of which fulfills the relevance criteria. This article by Christakis and Bird [5], is in turn referenced by another work we consider relevant, the article by Rachow et al. [18]. Only iterations during which another relevant articles were found are shown, i.e. applying the snowballing method to the articles [2], [18] and [21] did not produce a result.

Table 3.3: Results of the snowballing method.

Reference	Direction	Number of citations	Relevant articles
[11]	Backward	23	[2], [21]
	Forward	6	[10]
[10]	Backward	23	[20]
	Forward	0	-
[20]	Backward	43	-
	Forward	28	[5]
[5]	Backward	52	-
	Forward	34	[18]

As a result of the literature study six relevant articles were identified. Table 3.4 presents a brief summary of each article. The article by Martínez-Fernández et al. [10] is excluded from the list as it only provides an extension of the Q-Rapids model described in [11] and has the same first author. The article by Bergel et al. [2] was exchanged with a

more recent and more detailed article by Mordal-Manet et al. [12] depicting the Squale approach. This was necessary because a further investigation of the Squale model was not possible due to unavailability of the official website of the project and we had to rely solely on the information provided in the found article, which would have not been detailed enough.

Table 3.4: Overview of found articles.

Art- icle	Authors	Keywords	Summary	Rele- vant RQs
[11]	Martínez- Fernández et al.	Q-Rapids quality model, actionable an- alytics, rapid software development (RSD)	Quality model, which combines het- erogeneous data sources and sup- ports actionable analytics, imple- mentation and evaluation of the model	RQ1- RQ3
[12]	Mordal- Manet et al.	Squale quality model, software metrics, soft- ware quality	Industrially validated quality model based on aggregation of software metrics into higher level indicators	RQ1- RQ3
[21]	Wagner et al.	Quamoco quality model, product quality, quality assessment	Comprehensive quality model for closing the gap between abstract quality characteristics and concrete quality measurements	RQ1- RQ3
[20]	Sadowski et al.	Tricorder, Google, pro- gram analysis, static analysis	Platform for static analysis that is widely used by developers, im- plementation and evaluation of the platform	RQ1- RQ3
[5]	Christakis and Bird	Microsoft, program analysis, developers, empirical study	Empirical study of usage of program analysis tools among practitioners	RQ1
[18]	Rachow et al.	Clean Code, software quality, empirical study	Empirical study of what hinders the enforcement of code quality	RQ1

The first four articles ([11], [12], [21], [20]) describe four different approaches to software quality which were implemented and applied to real world projects and were evaluated in practice. The last two articles ([5] and [18]) depict empirical studies performed to examine developers' perspective on code quality and will not be considered during the

comparison in Section 3.4. These articles will be used later on to complement the findings of the interview-based study conducted for this work, which we describe in Chapter 4.

3.4 Comparison of Q-Rapids, Squale, Quamoco and Tricorder

In this section, we compare the approaches described in the articles [11], [12], [21], [20] to derive requirements and to identify design decisions which could be reused for our application. For better readability, we will hereon refer to the approach described in the article [11] as Q-Rapids approach, the approach described in the article [12] as Squale approach, the approach described in the article [21] as Quamoco approach and the approach described in the article [20] as Tricorder approach.

Table 3.5 contains information on the four approaches in a structured form to enable a focused comparison of the approaches.

Table 3.5: Comparison of the Q-Rapids, Squale, Quamoco and Tricorder approaches.

Q-Rapids	Squale	Quamoco	Tricorder
Used standard			
ISO/IEC 25010	ISO/IEC 9126	ISO/IEC 25010	-
Targeted roles			
All participants of the development process	All participants of the development process	All participants of the development process	Software developers
Approach description			
Raw data, possibly from heterogeneous data sources, is used to calculate the assessed metrics. Based on these metrics product and process factors are calculated. Then based on product and process factors quality aspects are calculated. A quality alert is issued, if a certain value is reached.	The model collects different raw measures and translates them into high-level marks in the [0;3] range. Marks are computed using continuous functions. Thresholds are set to determine what mark is acceptable or not.	Quality metrics are defined in the quality model and interpreted according to the utility functions defined in the model and then aggregated along the model's structure to derive utilities for individual quality aspects and for product's overall quality. Finally, the aggregated utilities are mapped onto more intuitive quality assessments.	Various frameworks and tools for program analysis ("analyzers") are integrated into one program analysis platform. Analyzer writers can create new and modify existing analyzers, which share a common API. The analysis is triggered at code review time and appears in form of warnings.
Used code-related metrics (RQ1)			

<ul style="list-style-type: none"> • Cyclomatic complexity per function • Density of comment lines per each file • Duplications • Number of quality rule violations per file, their severity and type • For each commit: files changed, lines of code added/modified/deleted, author, and revision • Number of passed tests • Duration of unit test execution • Test coverage 	<ul style="list-style-type: none"> • SLOC (Method Size and file) • Cyclomatic Complexity • Comment rate • Number of Methods per Class and other OO Metrics (coupling, cohesion, inheritance) • Violations of programming rules • Test coverage (line and branch coverage) • Number of javadoc 	<p>Metrics provided by:</p> <ul style="list-style-type: none"> • FindBugs • Gendarme • PMD • ConQAT • various other tools and manual instruments 	<p>Metrics provided by:</p> <ul style="list-style-type: none"> • ErrorProne • ClangTidy • Java Checkstyle linter • Pylint • AndroidLint • validators for project-specific schemas • other analyzers
---	--	---	--

Actionability of metrics (RQ2)

Raw data visualization to detect problematic parts of the program.	Focused drill-downs from traffic lights to problematic parts of source code.	Results can be used in a formative way to spot software defects in the source code.	Integration into development workflow, fixes are provided and can be applied without context change. Unnecessary information is avoided at all cost. Feedback loop: Analyzers with too many “not useful” warnings are removed from the platform.
Visualization of metrics (RQ3)			
Possibility to see the current value of the assessed metrics in the “Metrics”-View in a table form, a trend over time period as a graph	User-specific views on a dashboard	Possibility to zoom-in into a factor of a sunburst diagram and to compare different systems or versions of a program in a Kiviat diagram	Analysis results appear in code review as robocomments containing category of the result, location within code, error message, URL with details, list of fixes
Evaluation			
Evaluation of understandability and relevance of the model with 8 participants (one of them developer): Model was perceived as relevant but participants admitted difficulty in understanding normalized values.	-	8 semi-structured interviews with mostly developers. Model perceived as understandable but there was no agreement if model would improve productivity.	Evaluation of usability, code base impact, pluggability, scalability. In general, the developers were happy with usability. Number of violations went down over time, developers contributed plug-ins, scalability was also proven.

3.4.1 Used Standard

The Squale model is based on ISO/IEC 9126 and both Quamoco and Q-Rapids are based on ISO/IEC 25010, which replaced ISO/IEC 9126 in 2001. The Tricorder platform does not have any underlying norm.

3.4.2 Targeted Roles

The Tricorder approach is explicitly directed at software engineers, whereas all other approaches consider the quality needs of all participants of the development process.

3.4.3 Approach Description

There is one major difference that separates Tricorder from Q-Rapids, Squale and Quamoco. The main goal of Q-Rapids, Squale and Quamoco approaches is to provide a comprehensive model for an overall quality assessment of a software product. All three use measurements from analysis tools which they aggregate using numerical weights into high-level quality goals like maintainability.

These three models attempt to express the notion of quality without relying on discrete measurements. These three approaches also aim at making software programs written in different programming languages comparable to each other and even to estimate costs of non-quality [12]. Whereas these aspects present enormous interest for quality managers and product owners, the developers “prefer to interpret metric individual results rather than their aggregation to determine which component(s) of the project must be improved.” as the authors in [12] point out.

As opposed to these three approaches, Tricorder platform does not claim to provide a holistic picture of the software quality. Tricorder does not embed static analysis tools in an ISO standard based quality model like these other three approaches do. The main focus of Tricorder lies on encouraging developers to use static analysis tools for maintaining high code quality throughout the development process. The platform does not offer a systematic link to a quality model.

What all four approaches have in common, is considering project’s and company’s specificity in their work. The three ISO-based models and the Tricorder platform strongly emphasize the need for customization and extensibility of a quality model or tool.

3.4.4 Used Code-Related Metrics (RQ1)

Only the authors of the Q-Rapids approach explicitly mention the measures and metrics they used, therefore a direct comparison between models cannot be done.

The tool prototype of Q-Rapids uses the commercial tool SonarQube¹ as the source of its code quality measures.

Quamoco model accounts for 526 measures, which are provided by several different tools including FindBugs, Gendarme, PMD and ConQAT. Similarly, Tricorder also integrates a multitude of tools and frameworks for static analysis in its platform.

The metrics and measures mentioned by the authors of the Squale article also include those related to size, tests and programming rule violations, which seem to be used in all approaches. The authors also mention using object-oriented metrics like depth of inheritance and coupling between classes.

3.4.5 Actionability of Metrics (RQ2)

All four approaches claim to take developers' quality needs into account. We examine and compare the ways in which each approach ensures that the provided results are useful for developers in the sense that they are actionable, i.e. that these results can easily be transferred into tasks.

The three ISO-based models see their practical relevance for developers in the fact that the model enables a drill-down to a problematic part of the program which should be fixed. I.e. when a quality alert is issued, the product owner or quality manager informs the responsible developer and entrusts him or her with the issue. Software developers are not the main decision makers in such models. As the authors of Q-Rapids put it: "Code quality is actionable when the product owner decides to invest a cycle into maintainability and understandability." Developers can become part of the decision-making process if they are involved in the determining thresholds for a metric or in customizing the issuance of the quality alerts.

Tricorder handles the alerts in a very different way: quality warnings are issued during code review process, i.e. the analysis is integrated into development workflow. The issued warnings are made actionable through providing fixes, which can be applied without change of context by simply clicking *Apply Change*. Google recognized that

¹<https://www.sonarqube.org> (accessed May 8, 2019)

one of the main problems why developers are reluctant in trusting static analysis tools is high number of false positives. This was also the conclusion of several empirical studies such as [5] and [18]. One of the solutions to reduce the number of false positives is to introduce a feedback loop. For each warning the user can submit feedback as to whether the warning was useful. Analyzers providing too many “not useful” warnings are put on probation or deactivated altogether. This ensures that only analyzers considered useful are activated and the writers of the analyzers are motivated to provide especially readable warnings and helpful fixes with the warning.

3.4.6 Visualization of Metrics (RQ3)

In this subsection we examine the visualization mechanisms applied in the respective tool prototypes to display metrics.

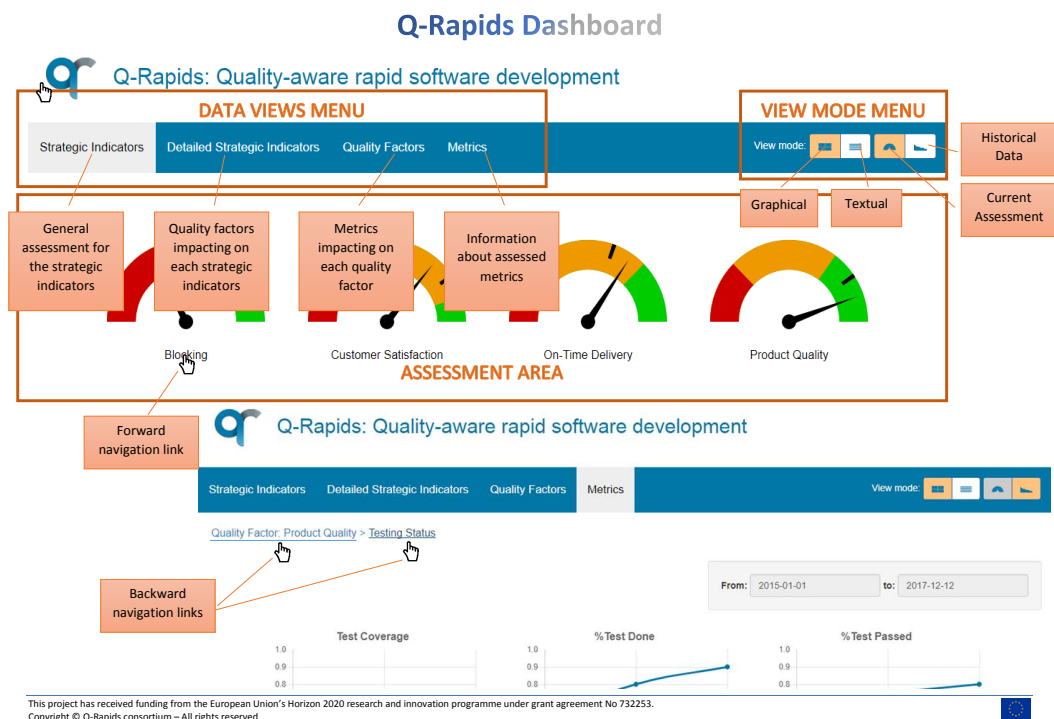


Figure 3.1: The dashboard of Q-Rapids tool.

In the Q-Rapids tool, there is a possibility to see the current value of the assessed metrics in the Metrics-View in a table form or let the dashboard show a trend over time period defined by user as a graph. The dashboard of the Q-Rapids tool is shown in Figure 3.1.

The official website of the Squale project is currently not available but in the article [2] it is said that quality is reported on specific dashboards and views, which are tailored for a specific user: programmer, project manager etc.

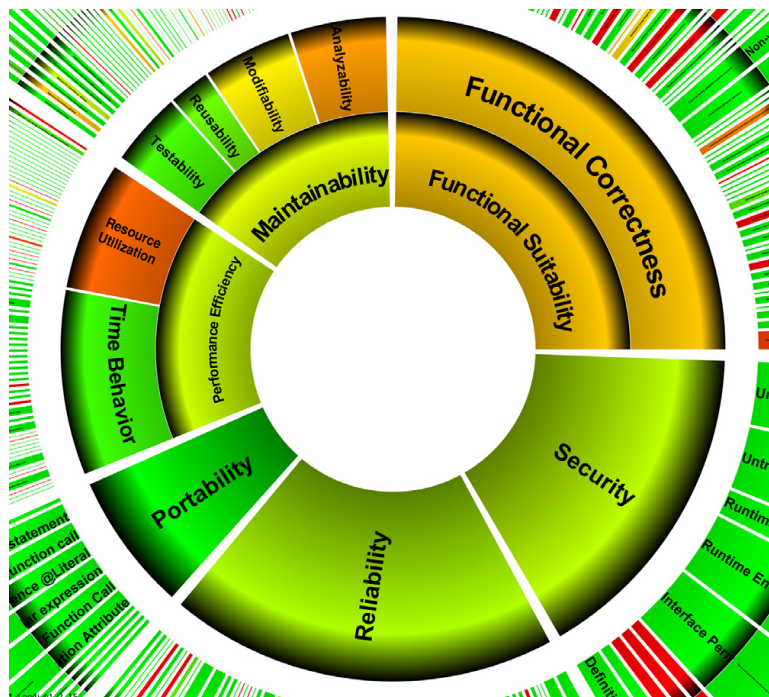


Figure 3.2: Quamoco tool: Sunburst visualization of quality factors.

The official website of the Quamoco project is also not available but the authors of the article provide examples depicted in Figure 3.2 and Figure 3.3. There is a possibility to zoom-in into a factor of a sunburst diagram and to compare different systems or versions of a program in a Kiviati diagram.

Analysis results of Tricorder appear in code review as robocomments containing category of the result, location within code, error message, URL with details and list of fixes. An example of such a robocomment is shown in Figure 3.4. No information was contained in the article about the possibility to track metric values over time.

3.4.7 Evaluation

The evaluation of understandability and relevance of the Q-Rapids model was conducted as an interview-based study with eight participants, only one of whom was a developer. Even though the model was perceived as relevant, interviewees admitted having difficulty understanding normalized values and would prefer actual values instead.

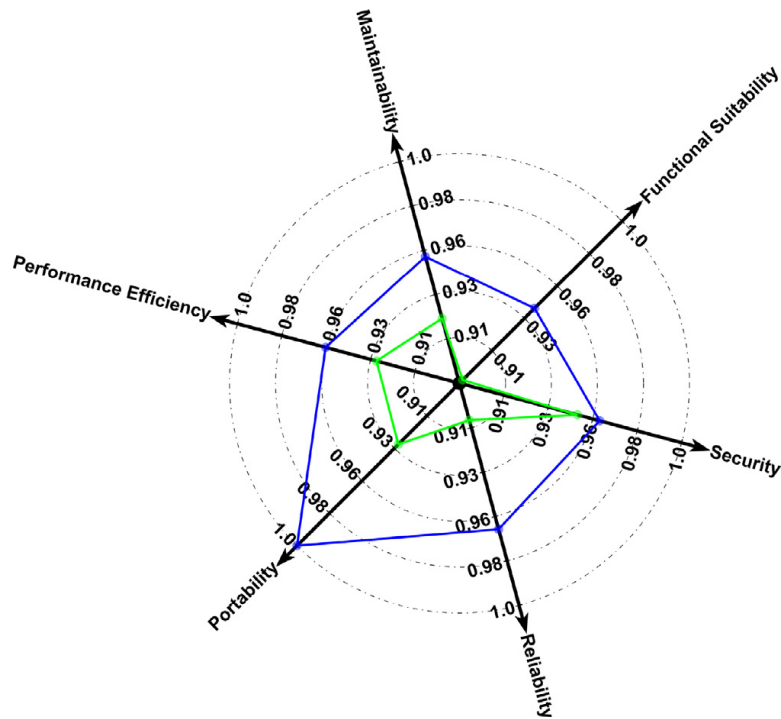


Figure 3.3: Quamoco tool: Kiviati diagram comparing two programs.

The authors of the Squale approach claim that the model is “well accepted by developers” but no details of the evaluation are provided in the article.

As to Quamoco approach, a study with eight semi-structured interviews with mostly developers as participants was carried out. The model seemed to be understandable for the interviewees but no agreement was reached if model would improve their productivity. The authors also point out that developing such a detailed model requires a lot of effort. They say: “We are not sure to what extent such effort can be expended in practical settings.”

At Google, evaluation of usability, code base impact, pluggability and scalability of Tricorder took place. Usability was measured by counting the not-useful click rates. The conclusion was drawn, that in general, the developers were happy with the results of the analysis platform. It was also observed that a reduction of the number of violations took place over time and an increasing number of pluggable analyzers were contributed by developers. Thus, the introduction of the platform alone motivated developers to write code with less violations and get involved in the developing of the analyzer services.

```
package com.google.devtools.staticanalysis;

public class Test {

  ▾ Lint      Missing a Javadoc comment.
  Java
  1:02 AM, Aug 21
  Please fix Not useful

  public boolean foo() {
    return getString() == "foo".toString();
  }

  ▾ ErrorProne String comparison using reference equality instead of value equality
  StringEquality
  1:03 AM, Aug 21
  (see http://code.google.com/p/error-prone/wiki/StringEquality)
  Please fix Not useful
  Suggested fix attached: show

  }

  public String getString() {
    return new String("foo");
  }
}
```

Figure 3.4: Example of Tricorder’s robocomment.

3.5 Answers to the Research Questions

In this section, we extract information from the synthesized articles relevant for answering the research questions RQ1 - RQ3.

3.5.1 Metrics Software Developers Need (RQ1)

All approaches integrate existing static analysis tools in their operationalized models to collect metrics. Even though all models were evaluated and considered useful by developers, there is no direct answer to the question which of the provided metrics developers considered most useful. The only approach which deals with this question is Tricorder with its integrated feedback loop and focus on configurable analyzers but even here no concrete metrics were named.

In all approaches, well-established program analysis tools are used (e.g. FindBugs and SonarQube) and all of them offer a huge variety of code quality metrics. But a large quantity of collected metrics does not automatically lead to a meaningful result: as Christakis and Bird [5] found out in their survey, the main pain-point for developers

when using program analysis tools was “wrong checks are on by default”, which confirms the intuition that not all metrics computed by program analysis tools are considered useful in the developer’s point of view.

This research question could not be sufficiently answered by comparing the presented approaches, which once more stresses the need for carrying out an empirical study to investigate the question of which metrics developers find most useful. We conducted such a study with eight software developers at SAP HANA and we describe the results in Chapter 4.

3.5.2 Transferring Metric into a Task (RQ2)

In the three ISO-based models metrics are made actionable through determining a threshold for every metric or a value based on an aggregation of several metrics. The analysis results of Tricorder are made actionable through an issued warning with a provided fix for a certain problem and the possibility to apply this fix without changing the application.

3.5.3 Visualization of a Metric (RQ3)

Following UI elements that can be used to display metrics were mentioned: a table containing all measures, a trend chart for the metric values for a user-defined time frame and a high-level diagram with a possibility to zoom-in on single values.

The visualization mechanism used by Tricorder is a pop-up window displaying the warning, together with a suggested solution for the problem and the button to submit feedback about the usefulness of the warning.

Whereas project managers and quality engineers are more interested in e.g. distribution of bugs among system modules and other high-level overviews, developers require finer-grained views that let them inspect analysis results for the specific parts of code they are working on. These views should be configurable.

3.6 Insights for This Work

This section focuses on requirements derived from the literature research and lists the design decisions of depicted approaches we intend to reuse for our prototype.

3.6.1 Stakeholder Interviews and Feedback Mechanism to Determine Useful Metrics

RQ1 could not be fully answered with examining of the four approaches. Therefore, the final selection of relevant metrics, which our prototype will display, will be carried out after the analysis of the results of the interviews we conduct with the future users: developers of SAP HANA.

To ensure that developers are not confronted with irrelevant information, Tricorder integrates a feedback loop to exclude analyzers from the platform that do not provide useful results. We also intend to provide a feedback mechanism for our prototype to ensure that the displayed information is actually useful for developers or at least enable the user of the application to configure the view in such a way that only relevant information is displayed.

3.6.2 Actionable through Thresholds and Drill-Down to Code

But solely displaying metrics is not enough. Our application should provide meaningful and actionable results that are helpful for programmers in their day-to-day work. Therefore, an important design decision that we will adopt from the ISO-based models as a requirement for our application are customizable thresholds for the metrics to be displayed.

For displayed metrics to be understandable there should always be a possibility to view the raw measurement behind this metric and easily arrive at the respective part of code. Granularity on file level should be ensured just like in Q-Rapids tool.

If metrics or measurements are combined to a single quality indicator, the aggregation should not be averaging out problematic aspects but stress them, just as the authors of Squale suggest. In the interviews we conduct, the question of how useful aggregated metrics are for developers will be addressed to help us decide if we should display aggregated values in our application at all.

The Tricorder system allows applying suggested fixes without context change but this functionality will not be implemented for this work as it is not technically feasible.

3.6.3 Visualization and Customization

The requirements for the visualization mechanisms of our prototype will have to be complemented with the results of the interview study, in which we will ask future users about their preferences regarding a meaningful metrics visualization.

From described approaches we would like to reuse the trend chart from Q-Rapids, where the user can trace the changes over time and the table form view, for the purposes of being able to jump to a specific part of the program.

Another important insight is enabling the user to customize the application. With configurable filters and dedicated views it can be possible to provide focused information to the user about the progress he or she is doing without being overwhelmed with the information that is not considered meaningful.

An important aspect ensuring customization for a program analysis tool is to allow easy addition of new metrics. Thus, extensibility or pluggability should also be a requirement for our application, just like shown in Quamoco and Tricorder approaches.

4 Interview Study

Since the focus of our work lies on the quality needs of software developers, we conducted an interview-based case study to complement the literature study described in Chapter 3. The overall goal of this study was to gain insights into the usage of metrics in the industrial context and reveal quality needs of the future users of our application - software developers at SAP HANA.

This chapter first describes the method used to design and perform the study and subsequently presents the results and gained insights for this work.

4.1 Method

In this section, we describe the preparation work for the study and interview questions we prepared for the participants.

4.1.1 Study Design

We decided to conduct a semi-structured interview study since a semi-structured interview is a widely used technique to collect relevant data in software engineering case studies [19]. The same research questions as listed in 3.1 were used to plan the study and derive the interview questions.

4.1.2 Planning and Preparation of Interviews

We structured the interview in three blocks. In the first block, we asked about currently used code quality metrics. In the second block, we presented a list of metrics we prepared before carrying out the interviews. This list was acquired during the initial literature study and is presented in Chapter 2. For each metric on the list, we discussed

its usefulness for the interviewee. The third block contained questions about the visualization of metrics. In the following, the interview questions are listed together with the motivation behind them.

Currently used metrics

By asking questions about currently used metrics, we wanted to identify metrics which have already proven themselves useful to developers. This contributes to answering RQ1. If useful metrics already existed, we wanted to find out how the interviewed developers benefit from these metrics in their daily work, which would also contribute to answering RQ2.

The block was comprised of the following interview questions (IQ):

IQ1: Which of the available metrics do you check regularly?

To receive more detailed answers and identify typical usage scenarios of the existing visualization tools for code quality metrics described in 2.2.2, the interviewees were seated in front of the monitor displaying the dashboards of these tools and asked to show the functionalities they used regularly.

IQ2: What coverage type is most interesting for you?

The existing platform for test coverage metrics was described in 2.2.2. We asked about which one of the provided coverage types (full, unit or diff) interests the interviewee most in his daily work.

IQ3: Do you use any external tools and plug-ins for your IDE to assess quality of your code?

We asked if an IDE was used and if so, whether special plug-ins for metric generation were used besides those provided out-of-the-box. These questions served the purpose of identifying metrics that developers were maybe missing in their provided development environment.

IQ4: What features do you miss in the current system?

We were aware that the answers to these questions would be difficult to summarize or to derive requirements from. Nonetheless, the questions served the purpose of warming-up the developers to a discussion about a code quality monitoring system.

Usefulness of specific metrics

The questions IQ5-IQ7 form the main part of the interview and directly address RQ1: *Which metrics do software developers need?*

IQ5: List of suggested metrics

As we could not expect the interviewees to name metrics on their own, we provided a list of existing metrics, which we prepared beforehand. Chapter 2 explains what served as the basis for this list and Table 2.1 lists the selected code quality metrics and categories of metrics presented to the interviewees.

The list of metrics served as a starting point for a discussion about which metrics software developers consider useful and why.

Before discussing if a metric was considered useful or not, we always first ensured that the interviewee knows the metric and if not, it was briefly explained and an example was presented to illustrate how the metric in question was calculated.

IQ6: Combination of metrics

We wanted to know if there exists a certain combination of metrics the interviewee considers useful. We brought in an example of how low coverage percentage combined with a high number of cyclomatic complexity may be indicating error-proneness.

IQ7: Metrics suggested by the interviewee

We concluded this block by asking, if there was a metric which might be useful in the daily work of a developer but was not on the provided list.

Visualization of metrics

In the last block of the interview (IQ8-IQ11), we aimed at discovering visualization mechanisms considered useful, which would help to answer RQ3, above all. Additionally, we put a special focus on how to make the visualization of metrics actionable, which would contribute to answering RQ2.

IQ8: Main function of a metric

This question was of a general nature and served as a warming-up to a discussion about an attractive metrics representation. In his book *Software development metrics* [13], Dave Nicolette mentions three functions of a metric: providing information (informational function), indicating a problem (diagnostic function) and influencing behavior

(motivational function). We asked the participant to tell us which aspect he considered most important. To this purpose, we provided a 5-point Likert scale to rate each of the three functions.

IQ9: Absolute value or traffic lights?

One of the questions in this block was whether the interviewee would prefer an absolute value or a traffic light visualization. This question was a follow-up to the question about the main function of a metric. After thinking about the effect metrics have on their consumers, we wanted to identify what developers would prefer - solely to be informed or to receive a warning and what would motivate them more.

IQ10: Aggregation of values

This question resulted from the criticism expressed towards the Q-Rapids tool [11] where the participants said they preferred seeing actual measurements instead of aggregations. To illustrate, the figure from the article describing Q-Rapids approach was shown to the participants.

IQ11: Spot - range - compare

The last question about visualization required rating these three common visualization methods: spot (value at the current moment), range (a trend over a certain period) and compare (comparison of values between two points in time) with a 5-point Likert scale. This question served to identify preferences of the users, so that we would know e.g. which view should be default view and which view should be optional.

4.1.3 Data Collection

The participants of the study were found through personal contacts. All study participants were male, with at least 6 years of professional experience in software development. The interviews were conducted in person, the duration of each interview was one hour on average. The answers to the interview questions were written down during the interview as recording was not allowed. All study participants were guaranteed that their names would not be mentioned in this work.

4.2 Results and Discussion

In this section, answers to the interview questions are presented and discussed with respect to deriving requirements for the web application.

4.2.1 Answers to the Interview Questions

In this subsection, the answers to the interview questions of the participated developers are described.

Currently used metrics

This subsection presents answers to the questions about the usage of the current tools dedicated to display metrics on code quality.

IQ1: Which of the available metrics do you check regularly?

All of the participants were interested in the coverage percentage (either displayed in the provided platform or in their own tools). The most common usage scenario of the current tool was checking coverage after submitting newly written code.

Other mentioned metrics checked regularly or sporadically referred less to intrinsic code quality but more to the organizational issues, such as e.g. who worked at the file last to clarify questions with him or her about the code or is the number of reported defects in code currently higher than the threshold appointed by the management.

IQ2: What coverage type is most interesting for you?

Seven out of eight participants named unit test coverage as most interesting coverage type for them. Half of the interviewees said that the full coverage was useful for them. Only three participants said they were interested in coverage of newly added code. One of the participants added that he checks diff coverage occasionally, e.g. after an effort was made to increase coverage.

IQ3: Do you use any external tools and plug-ins for your IDE to assess quality of your code?

One developer said that he uses a static analysis tool from the command line (as the tool is not integrated in his IDE) to generate metrics such as number of LOC and cyclomatic complexity. Another developer installed a plug-in for his IDE to check the C++ specific rules. The other six participants use the out-of-the-box functionalities of their IDE and obligatory checks automatically performed during the review phase as described in 2.2.1.

IQ4: What features do you miss in the current system?

As to the question about a missing feature some of the developers named branch (conditional) coverage besides the already provided line coverage.

Usefulness of specific metrics

IQ5: Suggested metrics

In Table 4.1, the numbers of replies to the IQ5 are presented.

Table 4.1: Overview of the answers to the interview questions on the usefulness of metrics.

Suggested Metric	Useful	Not Useful	Not sure
Number of LOC (for a file)	6	2	-
Number of LOC (for a method)	7	1	-
Comment percentage	-	8	-
Cyclomatic complexity	6	2	-
Halstead metrics	-	2	6
Frequently changed files	2	1	5
Files changed together	2	3	3
Number of authors (questionable)	3	4	1
OO metrics (mostly coupling)	7	-	1
Duplicate percentage	6	1	1
Number of code style violations	4	1	3

As can be seen from these results, comment percentage was unanimously considered as a metric without any predictive powers. Number of lines of code, on the contrary, was considered important, especially on the method level. Whereas large files were associated with long compile time and not necessarily an indicator of bad code quality, large methods were generally considered an indicator of low readability and maintainability.

Cyclomatic complexity was considered useful in general but also aroused a lot of criticism: half of the interviewees addressed the missing consideration of the nesting depth. Some of the participants pointed out that a long switch-case statement is considered more readable than a mixture of if-else statements, for and while loops. However, the number of cyclomatic complexity does not reflect it, as it only counts the binary decisions, thus assigning the method with a switch-case statement an equally high complexity.

Halstead metrics were least known by all developers. The metrics were briefly explained but were not considered useful. Some of the participants said that maybe if they had experience with working with it and get a feeling for the metric, maybe then they would

be able to assess its usefulness.

The metrics based on version control history did not arouse much interest either. This could be explained with the unfamiliarity with this category of metrics.

As to the object-oriented metrics, there was a clear tendency to consider metrics reflecting cohesion and coherence of classes more interesting than the metrics referring to the inheritance hierarchy. Even though most of the developers considered these metrics interesting, they were still not sure as to what they would do with this information, i.e. if they would act on it. Here again, this uncertainty could result from the lack of experience with certain metrics.

Most of the developers were familiar with duplicate percentage metric, because it was provided by the static analysis tool used during obligatory code review. Even though majority of them considered it useful in general, several participants pointed out that this warning usually was associated with files containing unit test cases, where duplicates are allowed and even desired. Some of the developers also pointed out that detecting code duplication would be more useful not just within a file but within a module.

As to the number of code style violations, most of the developers said that defining a team-specific code style rule set and then running an auto-formatting checker before submitting the code is certainly useful, but for this there should exist a consistent code style rule set to begin with.

What also became clear during this discussion is that developers usually know where the “Big Ball of Mud” is and, in most cases, they do not need metrics to identify these problematic parts of code, which is usually older legacy code. This older code produces a lot of warnings, which tend to annoy the developers. Refactoring a method with a very high number of complexity or LOC requires additional resources, which developers usually do not have. Some of them admitted, that they would gladly re-structure the older code to make it more readable and maintainable if there would be time budget for refactoring tasks.

IQ6: Combination of metrics

Some of the developers agreed with the provided example that low coverage combined with high cyclomatic complexity is an indicator for bad code, even though they added that low coverage is actually a good enough indicator by itself. One of the developers named a combination of duplicate percentage and cyclomatic complexity. Another participant pointed towards the question of weighting of combined values, which may be difficult to decide. One of the developers pointed out the importance of less abstract,

simple metrics.

IQ7: Metrics suggested by the interviewee

Following metrics were named as a response to this question or in the course of the interview (branch coverage and nesting depth were mentioned by several interviewees):

1. Recognition and warning of dangerous code structures in C++ such as raw pointer
2. Commit length (Motivation: when a commit exceeds a certain size, it is very probable there will be errors in the committed code)
3. Complexity measure that considers nesting depth
4. Duration how long a defect report is open
5. Suggestions (ideally by the IDE) for method extraction
6. Recognition of unused library dependencies
7. Dead code detection
8. Possibility to view all TODOs that exist in the comments in a list
9. Performance metrics (at the moment one just gets a notification if the code causes low performance after the merge into the main branch)
10. Custom component-specific settings, e.g. for error handling

Visualization of metrics

Of those, who use the existing tool, the overview containing e.g. the number of defects, coverage etc. is considered very useful (if only to be accountable for managers) but the possibility to drill-down to single values was always considered even more important.

The existing trend chart for test coverage was considered useful by the half of the participants, two of whom even emphasized its motivational aspect.

As possibility to make a metric actionable, following elements were mentioned:

1. Sorting function (e.g. sort files by their coverage percentage in ascending order to identify the so called “low hanging fruit”)

2. Coloring the metric value in red or green if a threshold exists

As to the question IQ8 - which aspect of a metric (informational, diagnostic or motivational) is more important - no discernible trend was discovered.

As to the question if a metric should be presented as a traffic light or as an exact value (IQ9), one person said that it depends on the metric. Five said they would rather have both (traffic light as entry point), and two said they would like only the exact value to be displayed.

Aggregation was considered useful only as an entry point by most of the developers (IQ10). In the answers to this question, the emerging trend was towards simpler values rather than aggregated indexes.

As to the spot-range-compare (IQ11), the results are not very clear: spot is considered most important, range is slightly higher than compare, but all the results were not far apart.

One of the developers suggested introducing an element of gamification to the entire code quality assessment process. He suggested to use a „score“, which gets higher each time you improve your code, with e.g. extracting a method.

4.2.2 Answers to the Research Questions

The notes from the interviews were examined with respect to answering our research questions. We pursued a less formal form of qualitative data analysis - the immersion approach [19], as due to the low number of participants in the study no statistically significant conclusions could be drawn.

The described blocks of IQs served as codes for the analysis. Additionally, we identified two emerging topics that were not directly addressed in our questions: excluding certain parts of the code from analysis and customization to team-specific quality needs.

Metrics software developers need (RQ1)

Based on the results of the interview study, we created a prioritized list of the metrics the interviewed developers consider useful for their daily work:

1. LOC (method)

2. Object-oriented metrics (mostly coupling)
3. LOC (file)
4. Duplicate percentage
5. Cyclomatic complexity

Transferring metric into a task (RQ2)

Confirming the results of the literature study, defining thresholds for a metric was considered helpful with respect to actionability. Another identified mechanism was sorting the files by the metric to detect problematic code.

Visualization of a metric (RQ3)

Developers seem to prefer absolute values to combinations and aggregations of metrics. Warnings or traffic lights are only useful as entry point, there always must be a possibility to view the single values.

4.2.3 Insights for This Work

From the answers presented above, we attempt to derive requirements for our application.

As an important result of the interview study, we identified the need for the possibility of filtering out specific parts of code during code quality assessment. Why would developers want to exclude parts of code from the analysis? When we came to discussing the metric *Cyclomatic Complexity*, several developers pointed out that they primarily discover complex methods in older legacy code and tend to ignore the warning as refactoring this code would not only take a large amount of time but has also a high probability of introducing errors. As developers explained, if the metric is calculated on the grounds of both older, “untouchable” code and their own code, then the result will de-motivate. They would rather see the analysis results exclusively for their newly written code. Additionally, the metric *Duplicate Percentage* was considered not useful, when applied to test code where duplicates are not necessarily indicating bad code. I.e. having the ability to exclude certain files from the analysis would allow a more focused view on

own code quality.

This result also confirms the conclusion made by Christakis and Bird during an empirical study at Microsoft [5]. As one of the major pain points when using program analysis, the participants of their study named “wrong checks are on by default”. In another study by Rachow et al. about Clean Code practices, the participants stated that “too many warnings” discourage developers from using an analysis tool [18].

We also identified the need for team-specific customization of the view. For some teams, a metric can be relevant, for the others not. And again, for the purpose of not annoying developers with high number of false positives or, simply put, information that they do not need, there should be a possibility to hide a metric, similarly to the Google’s platform Tricorder [20], where an analyzer is deactivated if considered not useful.

Due to the time restriction, we decided to only provide two metrics in our prototype: number of LOC and unit test coverage, as the data for these two metrics was already available and easily consumable from the system. Investing an effort in data pre- and postprocessing and extraction would go beyond the scope of this work. At the same time, we aim at designing the web application prototype in such a way, that adding a new metric can be achieved with low development effort. To this purpose, the non-functional requirement *Reusability* was identified.

5 Requirements

This chapter depicts requirements for our quality web application. We adopted the task-driven approach TORE [16] to specify the requirements: For our user role (persona), we identified the relevant tasks (user tasks) and refined them with saubtasks to identify the responsibilities of the system (system functions).

Section 5.1 describes a typical end user of the application. Section 5.2 lists all functional requirements as system functions and corresponding user stories. Section 5.3 names the non-functional requirements for the tool prototype. The User Interface Structure Diagram is presented in Section 5.4. Section 5.5 concludes the chapter.

The requirements were derived from the literature study described in Chapter 3 and from the interview study depicted in Chapter 4.

5.1 Persona

In the following, a fictitious persona is described. This persona represents a prototypical user of the application. The quality web application should support this user in his or her daily work. Describing the user's typical frustrations and needs should help the developers of the web application to slip into user's shoes and not to lose focus on user's needs throughout the entire development process.

Table 5.1: Persona: Software developer.

Job	Software developer
Biographic Facts	35 years old, male, holds a master's degree in Computer Science. Has a full-time job as a C++ developer in a large corporation.
Knowledge	Proficiency in C++, familiar with agile methodology, uses Visual Studio Code as his editor. Sometimes uses Cppcheck for static analysis.

Needs	Wants to see continuous improvement of his code in form of e.g. higher unit test coverage and reduction of McCabe's complexity of methods he works on.
Frustrations	Wants to monitor metrics only for the code he and his team is currently working on but the metrics his current tool is showing are calculated on the basis of the entire code base including the older legacy code.
Ideal features	Wants to decide for himself, what threshold a certain metric should have.

5.2 Functional Requirements

This section describes the requirements for the functionality of the quality web application. The defined user task (UT) describes the task at which the application supports the user. The user task is specified in more detail through subtasks (ST), system functions (SF) and user stories.

5.2.1 Tasks of the User

UT1 - Manage metrics

Software developers want to monitor code quality metrics. They want to determine thresholds for certain metrics. They want also to view metrics according to their preferences. They want to identify files which may need to be refactored.

ST1 - View metrics

By default, the metrics for the current date are shown. Software developers determine the order in which the files are shown. They search for specific files. They view metrics for files grouped by a certain criterion.

ST2 - Configure threshold for metric

Software developers determine which value of the metric is considered "good" and which value indicates that specific files should be looked at.

ST3 - Configure filter for files

Software developers save custom filter, so that they do not need to apply the desired

filters every time they check the metrics with the quality web application.

5.2.2 System Functions

The following section depicts the system functions describing the defined user tasks from system's perspective. To this purpose, the input and the output in the respective work space (WS) of the user interface are defined as well as the pre- and the postconditions of the system. Additionally, for each system function the role and the motivation of the user is described in the form of a user story.

Show current metric value

As a user, I want to be able to view current value of metric(s) for individual files. Table 5.2 describes the system function for this requirement. The system function supports subtask ST1.

Table 5.2: Show current metric value.

Preconditions (system)	At least one file with metrics exist
Preconditions (UI)	The URL of Quality Page is entered in the browser input field
Postconditions (system)	Nothing changed
Expected output (UI)	WS1: Table View, files with their current metric value and indicated threshold are shown

Filter files

As a user, I want to be able to filter files to check metrics for only these files. For example, I want to exclude test files from my view. Table 5.3 describes the system function for this requirement. The system function supports subtask ST1.

Table 5.3: Filter files.

Preconditions (system)	Files that meet the filtering criteria exist
Preconditions (UI)	WS1: Table View, from the drop-down menu of a desired filter, at least one criteria (e.g. name of the component or a code group) is selected
Postconditions (system)	Nothing changed
Expected output (UI)	WS1: Table View, only files that meet the filtering criteria are shown
Rules	If the files are filtered by component name, the output comprises all file names that belong to this component. If the files are filtered by custom group name(s), the output comprises all file names that belong to the group or groups (multiple group names can be selected).

Sort files

As a user, I want to be able to sort files by the metric value, e.g. when I want to check which files have the lowest ratio of unit test coverage or the highest number of lines of code. Table 5.4 describes the system function for this requirement. The system function supports subtask ST1.

Table 5.4: Sort files.

Preconditions (system)	Files with the calculated metric exist
Preconditions (UI)	WS1: Table View, in the column showing the metric, ascending or descending order is selected
Postconditions (system)	Nothing changed
Expected output (UI)	WS1: Table View, files are sorted according to the selected order
Rules	If the files are sorted in ascending order by their metric value, the output comprises all file names with those on top of the list, which metric values are the lowest. If the files are sorted in descending order by their metric value, the output comprises all file names with those on top of the list, which metric values are the highest.

Search files

As a user, I want to be able to search for a specific file. Table 5.5 describes the system function for this requirement. The system function supports subtask ST1.

Table 5.5: Search files.

Preconditions (system)	Files with the calculated metric exist
Preconditions (UI)	WS1: Table View, in the column showing the file names, a file name is entered in the search menu
Postconditions (system)	Nothing changed
Expected output (UI)	WS1: Table View, the file table only shows the file with the entered name
Rule	If a file with entered name does not exist, empty table is shown.

Hide/unhide a metric

As a user, I want to be able to hide columns with metrics that I do not want to monitor. After hiding a metric, I want to be able to view it again. Table 5.6 describes the system function for this requirement. The system function supports subtask ST1.

Table 5.6: Hide/unhide a metric.

Preconditions (system)	Files with calculated metric exist
Preconditions (UI)	WS1: Table View, metric that should be ignored is selected
Postconditions (system)	Nothing changed
Expected output (UI)	WS1: Table View, the column with selected metric is no longer shown

Show trend view

As a user, I want to be able to view a trend over a period of time for a certain metric. Table 5.7 describes the system function for this requirement. The system function supports subtask ST1.

Table 5.7: Show trend view.

Preconditions (system)	Files with calculated metric exist, files for which the metric should be shown are selected
Preconditions (UI)	WS1: Table View, metric that should be shown in the trend view is selected
Postconditions (system)	Nothing changed
Expected output (UI)	WS2: Trend View, trend for the selected metric is shown
Exceptions	If no files were selected, trend is shown for all files

Create threshold for a metric

As a user, I want to be able to configure thresholds of metrics. Table 5.8 describes the system function for this requirement. The system function supports subtask ST2.

Table 5.8: Create threshold for a metric.

Preconditions (system)	Files with calculated metric exist
Preconditions (UI)	WS3: Configuration View, metric to be configured is selected
Postconditions (system)	Threshold is created
Expected output (UI)	WS3: Configuration View, no change; WS1: Table View, created threshold is shown

Update threshold for a metric

As a user, I want to be able to configure thresholds of metrics. Table 5.9 describes the system function for this requirement. The system function supports subtask ST2.

Table 5.9: Update threshold for a metric.

Preconditions (system)	Threshold for the metric to be configured exists
Preconditions (UI)	WS3: Configuration View, new threshold is entered
Postconditions (system)	New threshold value is saved
Expected output (UI)	WS3: Configuration View, no change; WS1: Table View, new threshold is shown

Save selected filters

As a user, I want to be able to save all the filters that I applied to the files. Table 5.10 describes the system function for this requirement. The system function supports subtask ST3.

Table 5.10: Save selected filters.

Preconditions (system)	Files with calculated metric exist
Preconditions (UI)	WS1: Table View, only filtered files are shown
Postconditions (system)	The filters are saved
Expected output (UI)	WS1: Table View, only filtered files are shown (no change)
Exceptions	If no files were selected, trend is shown for all files

Apply saved filters

As a user, I want to be able to apply saved filters to metrics view. Table 5.11 describes the system function for this requirement. The system function supports subtask ST3.

Table 5.11: Apply saved filters.

Preconditions (system)	Files with calculated metric exist, a filter exists
Preconditions (UI)	WS1: Table View, saved filter is passed to the application
Postconditions (system)	Nothing changed
Expected output (UI)	WS1: Table View, only files not excluded by the custom filter are shown

5.2.3 Domain Data

The domain data model depicted in Figure 5.1 illustrates the relation between files and metrics as well as files and file groups. Each file belongs to one or more file groups. Component Group represents all files of a specific component of the HANA code base such as e.g. *Calculation Engine*. A file always belongs to only one Component Group. A so-called Custom Group of files can be defined by a user, who, e.g. wants to group

his or her test files into one group and older legacy code files into another etc. A file can belong to only one custom group.

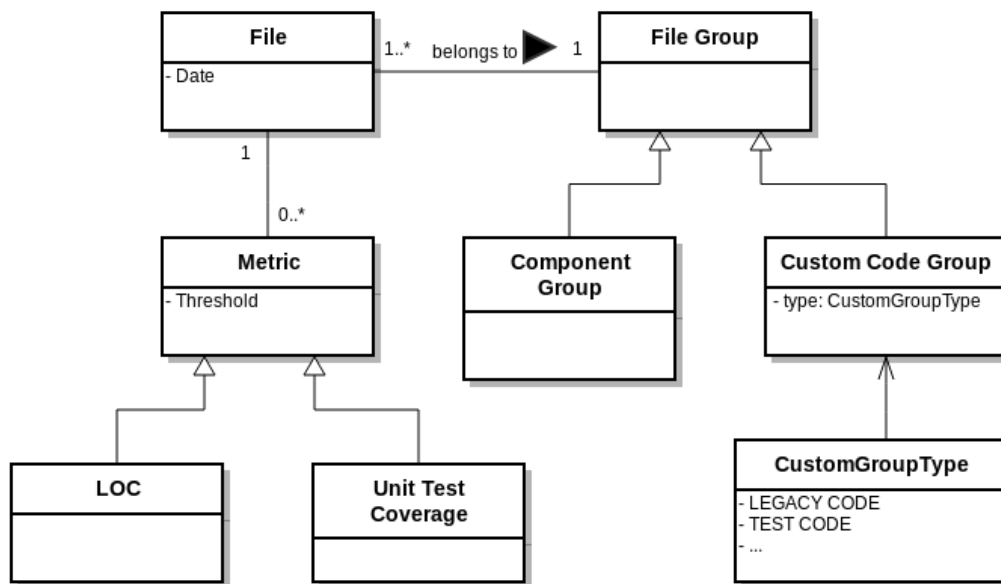


Figure 5.1: Domain Data Diagram.

5.3 Non-Functional Requirements

Non-functional requirements (NFR) can be used to describe how good certain functionalities of the system work. Tables 5.12 and 5.13 define the non-functional requirements. Categories of listed NFR are based on ISO/IEC25010 ¹. The named project phase refers to the phase of the development process during which this NFR has to be taken into account.

5.4 User Interface Structure Diagram

The User Interface Structure Diagram in Figure 5.2 depicts all necessary Work Spaces (WS) and possibilities to navigate between them. For each Work Space, all data needed for it and all system functions that can be triggered by the user from it are listed.

¹<https://iso25000.com/index.php/en/iso-25000-standards/iso-25010> (accessed June 10, 2019).

Table 5.12: NFR: Performance efficiency (Time-behavior)

Category	Performance efficiency (Time-behavior)
Description	The response time of the database to all SQL queries together with the frontend response time does not exceed five seconds.
Explanation	Five seconds response time due to the distributed execution of the SQL query on multiple hosts. Additionally, the table containing relevant information has more than one million entries. However, most users of a modern web application would not accept a waiting period much longer than that.
Project phase	Design and implementation

Table 5.13: NFR: Maintainability (Reusability)

Category	Maintainability (Reusability)
Description	System architecture easily allows adding new metrics to the application.
Project phase	Design and implementation

The WS1 (Table View) is the centerpiece of the application and is the default view user sees after he or she opens the URL of Quality Page in the browser. From it, the user can navigate to either WS2 (Trend View) to view the trend chart for a metric or to WS3 (Configuration View) to configure threshold for a metric.

5.5 Summary

In this chapter, we described a potential user of the application in the role of a software developer. One user task and three subtasks describe the tasks of this role. Ten functional requirements were elicited and presented each in form of a user story and a system function. Two non-functional requirements were identified and described. Table 5.14 contains all requirements defined in this chapter.

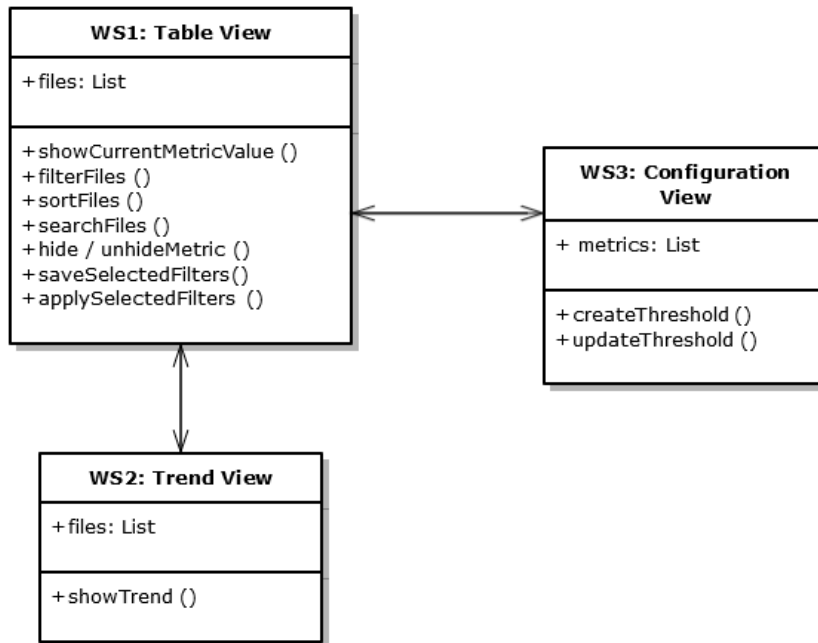


Figure 5.2: User Interface Structure Diagram.

Table 5.14: Overview of functional and non-functional requirements.

Id	Type	Name
1	SF	Show current metric value
2	SF	Filter files
3	SF	Sort files
4	SF	Search for files
5	SF	Hide / unhide a metric
6	SF	Show trend view
7	SF	Create threshold for a metric
8	SF	Update threshold for a metric
9	SF	Save selected filters
10	SF	Apply saved filters
11	NFR	Performance efficiency (Time-behaviour)
12	NFR	Maintainability (Reusability)

6 Design and Implementation

This chapter describes important design decisions when implementing both the functional and non-functional requirements defined in Chapter 5. We describe how we implemented the requirements and what challenges we dealt with during the implementation phase.

6.1 Architecture of the Web Application

In this section, we describe the general architecture of the Quality Page and the design decisions related to it.

6.1.1 Landing Page vs. IDE Plug-In

First, we address the question why the metrics visualization tool was designed as a landing page and not as a plug-in for the IDE. The upside of a system directly integrated into the IDE is obvious: a much better workflow integration would be achieved because no separate dashboard has to be accessed and a software developer could change code without any change of context.

Nevertheless, the decision was made to design an extra landing page as software developers at SAP HANA all use different text editors or IDEs. Therefore, a central tool has to be provided to support code quality management. One of the upsides of such a tool is the possibility to provide precalculated analysis results, making it faster than a system executed locally. In the context of the HANA code base with over 3.5 million lines of code, even switching to a different code branch would take very long, if done locally.

6.1.2 Frontend and Backend Technologies

A typical web application uses client-server model with a frontend-backend relationship. In our case, the backend application generates data in the JSON format and exposes it via a RESTful service. The Representational state transfer (REST) API end-point can be accessed via a URL using Hypertext transfer protocol (HTTP). The frontend application instantiates a JSON Model class and loads the data from the URL into the model.

In Figure 6.1, a deployment diagram shows the components of the system.

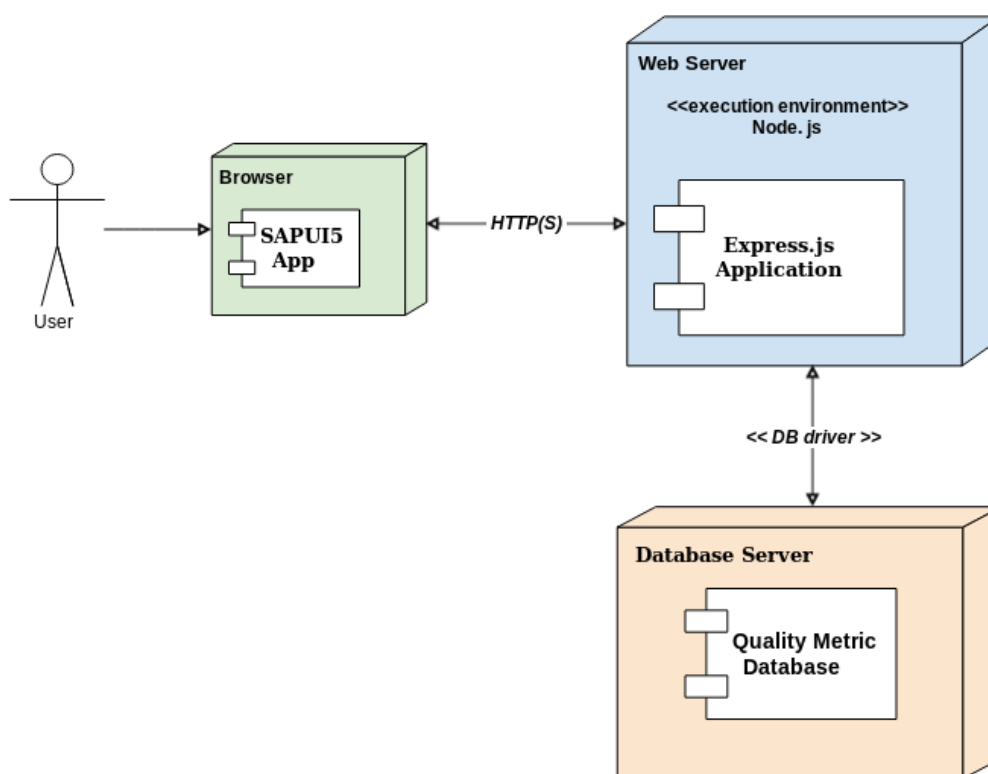


Figure 6.1: Deployment Diagram of QualityPage.

Frontend: SAPUI5

SAPUI5 is a UI framework developed by SAP SE and contains a set of libraries used to build responsive web applications. SAPUI5 follows the Model-View-Controller design pattern to create a separation between data (Model), business logic (Controller) and representation of data (View).

Backend: Node.js and Express.js

Node.js is a JavaScript runtime built on Google Chrome's V8 JavaScript Engine. With Node.js, JavaScript code can be run server-side. Express.js is a web framework written in JavaScript and hosted within the Node.js runtime environment. With Express, the data is retrieved from the database (called Quality Metric Database in Figure 6.1) with a SQL query and exposed using REST service to a specified URL. From this URL, the data is then consumed by the SAPUI5 frontend application.

6.2 Class Diagram

Figure 6.2 illustrates the entire class diagram of the frontend application. For reasons of readability, some details are left out but will be provided when we describe the implementation of the individual system functions.

In the following, the main files are described.

Main.view.xml

This file contains all the UI elements and fragments, which represent reusable parts of the UI, such as e.g. a trend chart or a table. In SAPUI5, independent UI elements such as Button or Label are called controls.

Main.controller.js

The controls specified in the XML files have an event listener attached. The events are handled by the methods defined in Main.controller.js.

manifest.json

This file contains the metadata description of the application. In the manifest.json models are described which are loaded upon starting the application. The i18n model handles translatable texts. Thus, translating the app in another language would not require changing the entire code but solely translating the content of the texts contained in the model. Other models are required for the content of the drop-down menus of the filters. The routing configuration is also specified in this file. This is necessary to handle navigating on the web page and updating the URL.

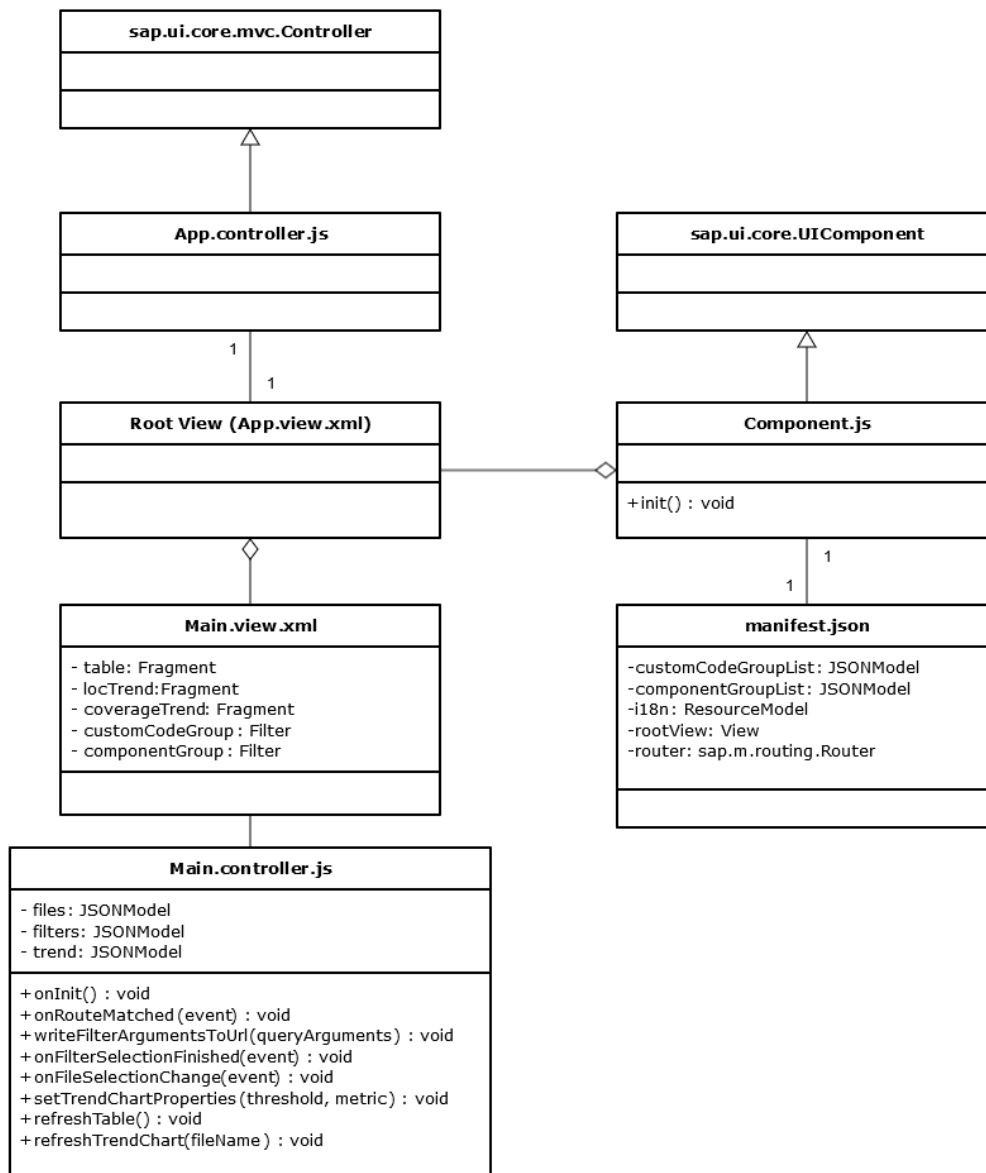


Figure 6.2: Class Diagram of the Frontend Application.

6.3 Functional Requirements

In this section, we describe how functional requirements were implemented. For the implementation of each system function, if necessary, a more detailed excerpt from the overall class diagram depicted in 6.2 is provided. Then, if there were any design issues to solve, we describe them and provide a decision tree generated with ConDec, a JIRA

plug-in for documentation and exploitation of decision knowledge¹.

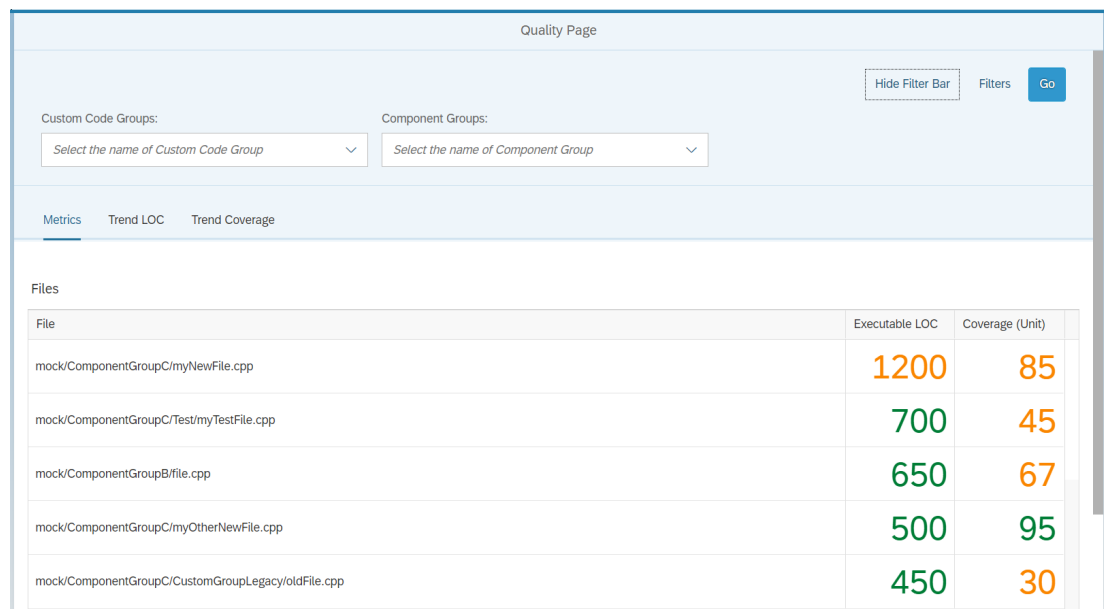
6.3.1 SF Show current metric value

The data to fill the table is provided by the backend. To this purpose, a database connection is initiated and, with a SQL query, data is retrieved from the database. This data is then exposed in JSON format at a specified end-point. In the front-end, a JSON Model is instantiated. Then, it loads the exposed data from the endpoint. The data is then rendered with the Table.fragment.xml.

For the first time the user accesses the page in the browser, in order not to display an empty table, we decided to limit the number of rows to 1000. Thus, the initial rendering of data takes less time. This decision addresses the non-functional requirement Performance Efficiency described in 5.12.

Before filtering and sorting of his own, the user is presented with files pre-sorted by the number of LOC (which happens in the backend). We sort the files by their number of LOC in descending order, as providing file names in alphabetic order does not present a useful use case. The user sees the largest files at first sight.

Figure 6.3 illustrates the default view when the user first accesses the application.



The screenshot shows a web interface titled "Quality Page". At the top right, there are buttons for "Hide Filter Bar", "Filters", and "Go". Below these are two dropdown menus: "Custom Code Groups:" and "Component Groups:", both with the placeholder text "Select the name of Custom Code Group" and "Select the name of Component Group" respectively. Below the dropdowns are three tabs: "Metrics", "Trend LOC", and "Trend Coverage". The "Metrics" tab is active. Below the tabs is a table titled "Files". The table has three columns: "File", "Executable LOC", and "Coverage (Unit)". The data is as follows:

File	Executable LOC	Coverage (Unit)
mock/ComponentGroupC/myNewFile.cpp	1200	85
mock/ComponentGroupC/Test/myTestFile.cpp	700	45
mock/ComponentGroupB/file.cpp	650	67
mock/ComponentGroupC/myOtherNewFile.cpp	500	95
mock/ComponentGroupC/CustomGroupLegacy/oldFile.cpp	450	30

Figure 6.3: Implementation of SF Show current metric value.

¹<https://github.com/ures-hub/ures-condec-jira> (accessed June 25, 2019)

6.3.2 SF Filter files

This system function is realized with two UI elements (controls): the `FilterBar` and the `MultiComboBox`. The `FilterBar` has a toolbar which is always visible, but the filter area can be hidden to reduce the space required. The user can also deactivate individual filters, thus also reducing the number of UI elements for a more focused view. The `MultiComboBox` control is integrated within the `FilterBar`. This control serves the purpose of providing the filter criteria for the user to choose from.

Files can either be filtered by the name of the HANA component they belong to (Component Group) or they can be filtered by the name of a user-specific group (Custom Code Group). With the first filter, we attempt to fulfill the requirement to have a project-specific view. With the second filter, we handle the wish to exclude certain files from the analysis (such as test code or older legacy code).

In Figure 6.4, the excerpt from the class diagram is depicted. Both `FilterBar` and `MultiComboBox` are defined in the `Main.view.xml`. First, the user selects one or more items from the drop-down menu of `MultiComboBox`. This drop-down menu is filled with the data retrieved from the models declared in the `manifest.json` file. An event is fired after the user selected one or more filter criteria and the event object is passed to the event handler method `onFilterSelectionChange`. The method extracts the selected items from the event object representing filter criteria. These filter criteria are passed to the backend as a string. The string is again passed to a SQL query of the database, where files are mapped to the selected code groups. The response of the database is loaded into the Files Model. The view representing Files model is refreshed and now shows only the files that meet filtering criteria.

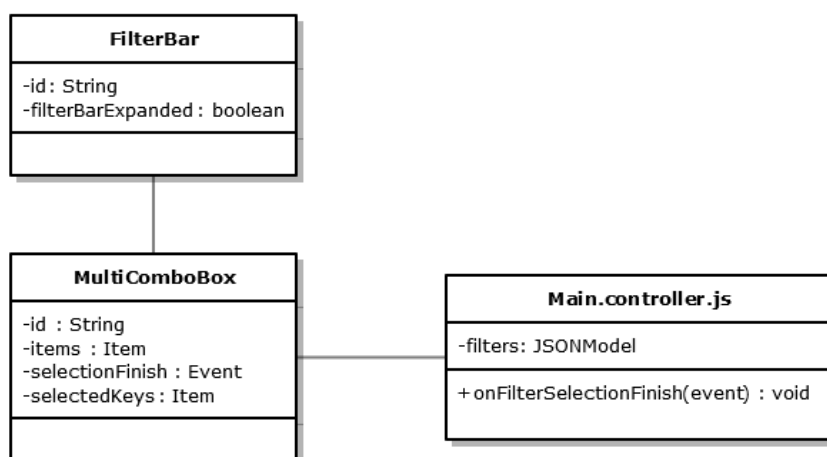


Figure 6.4: Class Diagram for the implementation of SF Filter files.

Figures 6.5 and 6.6 illustrate the implemented filters.

File	Executable LOC	Coverage (Unit)
mock/ComponentGroupC/myNewFile.cpp	1200	85
mock/ComponentGroupC/Test/myTestFile.cpp	700	45
mock/ComponentGroupB/file.cpp	650	67
mock/ComponentGroupC/myOtherNewFile.cpp	500	95
mock/ComponentGroupC/CustomGroupLegacy/oldFile.cpp	450	30

Figure 6.5: Implementation of SF Filter files for Custom Code Groups.

File	Executable LOC	Coverage (Unit)
mock/ComponentGroupB/file.cpp	650	67

Figure 6.6: Implementation of SF Filter files for Component Groups.

6.3.3 SF Sort files

The SAPUI5 framework already provides a built-in sorting functionality for columns of a table. We use it by setting the `sortProperty` attribute of the column as displayed in Figure 6.7.

The column header menu now provides two sort options: *Sort Ascending* and *Sort Descending* as shown in Figure 6.8. The user selects one of the options to sort the corresponding column accordingly.

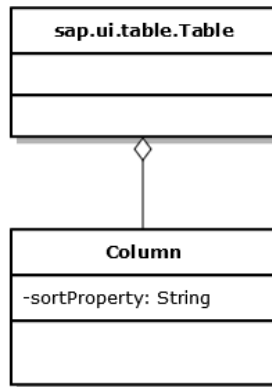


Figure 6.7: Class Diagram for the implementation of SF Sort files.

Executable LOC	Coverage (Unit)
1200	43
700	67
650	

Figure 6.8: Implementation of SF Sort files.

6.3.4 SF Search for files

The system function was implemented by defining the attribute `filterOperator` in the column with file names of the table, contained in the `Table.fragment.xml`. Figure 6.9 shows the class diagram.

For the class `sap.ui.table.Column`, there is a possibility to define a `filterOperator`, which is applied to the file names contained in the column. The options for the `filterOperator` include e.g. *Contains*, *EndsWith*, *StartsWith*. We chose the *Contains* operator, as it creates less restrictions. The file names usually follow this pattern:

“<componentName>/<subfolderName>*/<fileName>.<fileExtension>”.

The user can not be expected to know the names of all subfolders, neither can he or she be expected to remember if the file extension was `.cpp` or `.cc`, for C++ file, for example.

Table 6.10 illustrates the described decision knowledge.

The user can perform the search per right-click on Files column, entering the search string into the input field named Filter and pressing Enter.

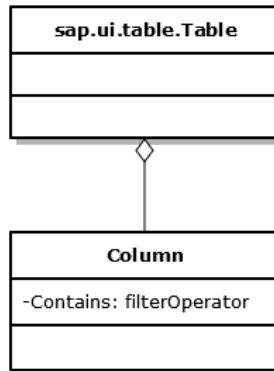


Figure 6.9: Class Diagram for the implementation of SF Search for Files.

6.3.5 SF Hide / unhide a metric

The SAPUI5 framework already provides a built-in functionality for hiding columns of a table. We use it by setting the `showColumnVisibilityMenu` property of the table to `true` as displayed in Figure 6.12.

The column header menu now provides the option **Columns** as shown in Figure 6.13. By clicking on the option, the user views the list with all column names and can select one or more columns by clicking on them. The column names with a check mark are visible. By clicking on the check mark, it disappears and the column is hidden.

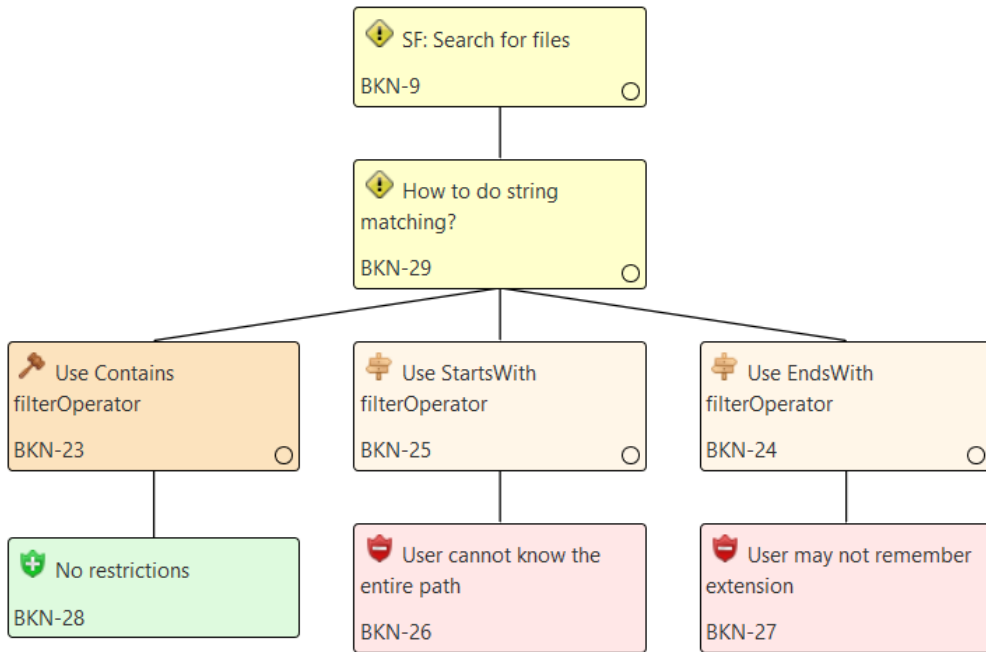


Figure 6.10: Decision knowledge for the Search for Files implementation.

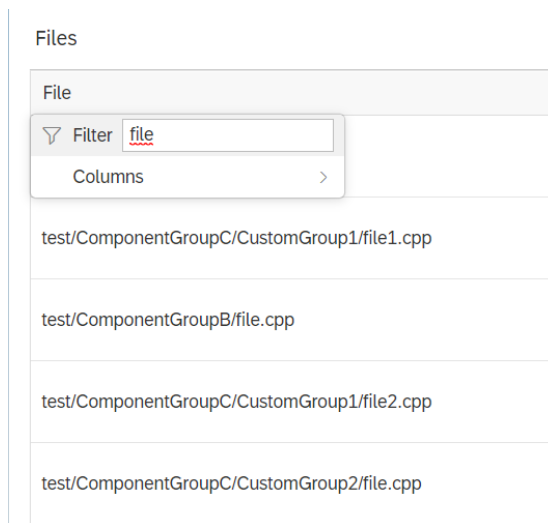


Figure 6.11: Implementation of SF Search for files.

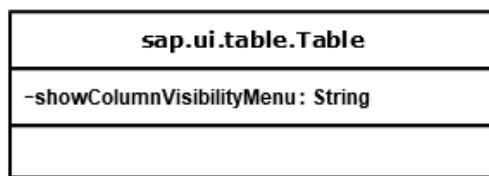


Figure 6.12: Class Diagram for the implementation of SF Hide / unhide a metric.

Executable LOC	Coverage (Unit)
700	85
650	87

Sort Ascending

Sort Descending

Columns >

✓ File

✓ Executable LOC

✓ Coverage (Unit)

Figure 6.13: Implementation of SF Hide / unhide a metric.

6.3.6 SF Show trend view

The invocation of this system function opens a new Work space (WS) – Trend View. Figure 6.14 shows the class diagram.

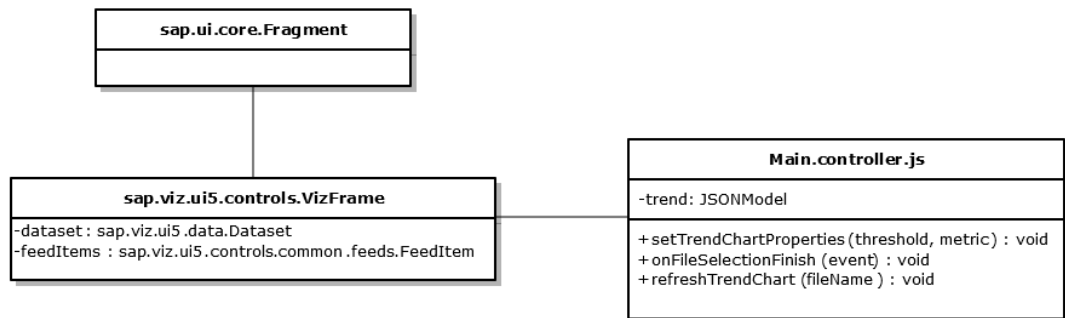


Figure 6.14: Class Diagram for the implementation of SF Show trend view.

Each metric's trend is realized within a separate fragment. The used control is VizFrame which receives data from the Trend Model. As soon as the user has selected a file, the event handler method passes the file name to the backend, the backend delivers the data at specified endpoint and the trend model is updated with the new data received from the backend.

The realization of the Trend View is illustrated in the Figure 6.15.

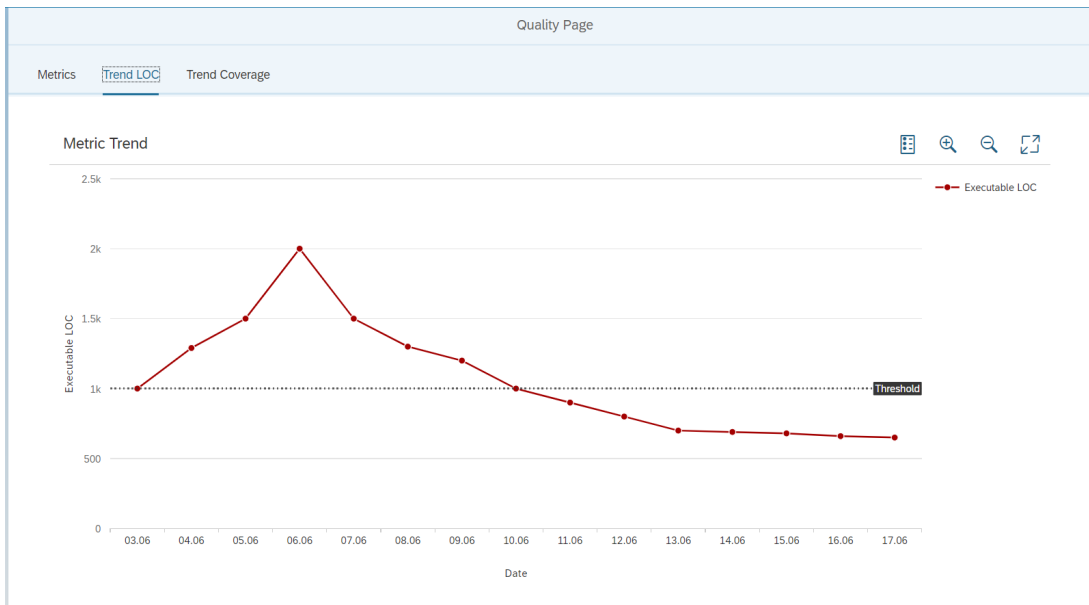


Figure 6.15: Implementation of SF Show trend view.

6.3.7 SF Create and update threshold for a metric

These two system functions could not be realized to the extent specified in the Chapter 5 due to time constraints. The threshold is provided as a fixed value in the code, but cannot be updated through user input, i.e. the Configuration View is not provided in the current implementation. In the Table view, the threshold is indicated with coloring the values which are considered *good* with green and those which are considered undesirable with orange color. To this purpose, we use the class `NumericContent`, where we pass a function to the attribute `valueColor`, determining color for each value in the column.

A possible implementation could be described as follows. On the frontend, the user clicks a button `Define threshold`, which opens a dialog, where the user selects a metric from a drop-down menu and, e.g. with a slider, determines the value for the threshold. Additionally, the user has to determine if values above the threshold are considered *good*, as is the case with coverage percentage or *bad*, which would be the case for the number of LOC. This input is passed to the backend and saved to the database.

6.3.8 SF Save selected filters

This system function was realized by passing the filter content to the URL. We extended the routing configuration: Instead of `/qualityPage/`, we allow the corresponding route to have the pattern: `qualityPage/##/?<name of filter>=<selected filter criteria(url en-`

coded)>.

An example would be: `qualityPage/#/?componentGroups=Component Group A, Custom Group B`.

We use the name of the filter, e.g. `componentGroups` as the URL parameter for the criteria which was selected in the filter (realized as described in 6.3.2). This parameter is passed to the router. We add an event listener for the matched event of the main view route. We keep the URL and the UI in sync by navigating to the current target again with the current value of the filter input field.

Each time new filter criteria are selected, the URL is updated and can be bookmarked by the browser.

6.3.9 SF Apply saved filters

To apply the saved filters, the user can simply open the bookmarked URL in his or her browser and the view will be provided with already applied filters.

6.4 Non-Functional Requirements

This section describes how we addressed the non-functional requirements for the Quality Page.

6.4.1 Performance

Following decisions were taken and implemented during the implementation phase to ensure the fulfillment of the NFR Performance efficiency (Time-behavior) described in 5.12.

Asynchronous loading

With enabling asynchronous loading, the runtime can load all the modules and preload files for all declared libraries asynchronously. The browser is capable of executing multiple requests in parallel, without blocking the UI thread.

Pre-Loading JSON Models

All models described in the manifest.json file are automatically instantiated when the application is started. Two of the models (the list of custom code groups and the list of component groups) do not need to be filtered and therefore can be loaded at the beginning, which saves time later on.

Filtering and Calculations in the Backend

Through filtering and pre-calculating some of the data in the backend, where it can be done more efficiently than in the frontend, overhead is reduced and time can be saved on the client side.

6.4.2 Reusability

Following measures were taken during implementation phase to ensure the fulfillment of the NFR Maintainability (Reusability) described in 5.13.

Modularization of UI

The Model-View-Controller design pattern of SAPUI5 already supports reusability. Additionally, we used **Fragments** – light-weight UI parts – to split up the views into reusable parts. When used within declarative Views, the Fragment content is imported and seamlessly integrated into the View. This way, we achieve modularization of UI without fragmenting the controller structure. When adding another metric, the trend chart fragments for existing metrics can be easily reused.

Steps to add a new metric

For the backend, a SQL query to the database has to be written to retrieve the necessary data and a new endpoint where the retrieved data should be exposed at, should be created. For the frontend, a new column for the metric in the existing Table control has to be added. For the trend chart, a new tab has to be created.

7 Quality Assurance

Unfortunately, the Quality Page could not have been applied to visualize the quality metrics for the code which implements the Quality Page. The required metrics are collected for the HANA database. However, this application is part of a larger tooling framework for which metrics are not collected in the same way as for HANA. Thus, the retrieval process of metric data was not applicable to the Quality Page. In the following, we describe how the quality assurance process for the Quality Page was performed.

Table 7.1 depicts manually extracted number of LOC for individual files of the frontend application of the Quality Page.

Table 7.1: Number of LOC per file.

File	Number of LOC
App.controller.js	6
Main.controller.js	141
i18nen.properties	9
App.view.xml	7
Main.view.xml	66
Table.fragment.xml	65
LocTrend.fragment.xml	46
CoverageTrend.fragment.xml	46
Component.js	12
index.html	43
manifest.js	67

7.1 Continuous Integration and Tests

A typical iteration for quality assurance for this project started with the developer submitting the code to the source code VCS, such as Gerrit Code Review¹. The VCS calls an automation server (Jenkins², for instance), which starts a voter job that trig-

¹<https://www.gerritcodereview.com> (accessed June 30, 2019)

²<https://jenkins.io> (accessed June 30, 2019)

gers static code checks to check the code style (with, e.g., ESLint³). Existing unit and integration tests are triggered automatically by a test runner (for example, Karma). The automation server gives immediate feedback. Subsequently, the developer assigns experts that should review the code. When the voter job and the human reviewer(s) both approved, the submitted change is merged into the main branch.

In this context, we distinguish between static and dynamic testing.

7.2 Static Tests

Static tests included, on the one hand, the static analysis performed by the code analyzer (*linter*), which checked whether the code follows the JavaScript code style conventions and, on the other hand, the code reviews conducted using the code collaboration tool integrated into the VCS. At least two reviewers were assigned and each of them submitted their feedback promptly. The review process ran parallel to the developing process, i.e. the developer was not blocked by ongoing reviews.

The functionalities of the code collaboration tool included the possibility to post comments to a specific line of code, an entire file or a commit message. All participants could reply to a comment, thus every change could be thoroughly discussed.

Before approval, the reviewers also dynamically tested the application on their machines by fetching the respective code version, running the application and performing system tests for the implemented functionality.

7.3 Dynamic Tests

7.3.1 Unit Tests

In SAPUI5, to test functionalities which are not placed on the screen and do not require user interaction, unit tests are written in Sinon.JS⁴. Most of the functionalities of our application could not be tested with unit tests.

³<https://eslint.org> (accessed June 30, 2019)

⁴<https://sinonjs.org> (accessed June 30, 2019)

7.3.2 System Tests with Mock Server

As the application was not well-testable on the level of unit tests, we performed extensive system tests with a mock server.

Mock server is a mocking framework that is used to simplify system testing and to decouple development teams by allowing to develop against a service that is not complete or unstable. Our mock server simulates the backend application and does not require a network connection since no queries to the actual database are sent. The mock server provides a “fake” response from the database.

In the following, a short description of test cases for the system functions specified in 5 is given. As the detailed description of pre- and postconditions for the system and for the user interface is given for each system function in 5.2.2, we limit the description to the performed test steps of the respective test cases and then present the actual result as seen on the UI.

Test Show current metric value

Table 7.2 documents the system test case performed to test the SF Show current metric value.

Table 7.2: System test for SF Show current metric value.

Test case	Test steps	Actual result
Test show current metric value	Open Quality Page in the browser.	File table contains correct file names, number of LOC and unit coverage percentage. Threshold is indicated according to the fixed value defined in the code.

Test Filter files

As the actual filtering is performed in the backend with the SQL query, the system function could not be fully tested with the mock server but with the actual backend server. Table 7.3 documents the system test cases performed to test the SF Filter files.

Table 7.3: System tests for SF Filter files.

Test case	Test steps	Actual result
Test filter files by the name of Custom Code Groups	1. Click on the filter of Custom Code Groups. 2. Select one or multiple names from the drop-down menu. 3. Click on Go button to complete the selection.	Files are filtered according to the filter criteria.
Test filter files by the name of Component Groups	1. Click on the filter of Component Groups. 2. Select one or multiple names from the drop-down menu. 3. Click on Go button to complete the selection.	Files are filtered according to the filter criteria.

Test Sort files

Table 7.4 documents the system test cases performed to test the SF Sort files.

Table 7.4: System tests for SF Sort files.

Test case	Test steps	Actual result
Test sort files by number of LOC in ascending order	1. Click on the header of the <i>Executable LOC</i> column. 2. Select Sort Ascending .	Files are sorted by number of LOC in ascending order.
Test sort files by number of LOC in descending order	1. Click on the header of the <i>Executable LOC</i> column. 2. Select Sort Descending .	Files are sorted by number of LOC in descending order.
Test sort files by coverage percentage in ascending order	1. Click on the header of the <i>Coverage (Unit)</i> column. 2. Select Sort Ascending .	Files are sorted by coverage percentage in ascending order.
Test sort files by coverage percentage in descending order	1. Click on the header of the <i>Coverage (Unit)</i> column. 2. Select Sort Descending .	Files are sorted by coverage percentage in descending order.

Test Search for files

Table 7.5 documents the system test cases performed to test the SF Search for files.

Table 7.5: System tests for SF Search for files.

Test case	Test steps	Actual result
Test search for existing files	1. Click on the header of the <i>File</i> column. 2. Enter a search string into the input field. 3. Press Enter.	File names containing the search string are shown.
Test search for non-existent files	1. Click on the header of the <i>File</i> column. 2. Enter a search string into the input field Filter. 3. Press Enter.	An empty table is shown.

Test Hide / unhide a metric

Table 7.6 documents the system test cases performed to test the SF Hide / unhide a metric. Both test cases were carried out for both metrics (the number of LOC and the unit test coverage).

Table 7.6: System tests for SF Hide / unhide a metric.

Test case	Test steps	Actual result
Test hide a metric	1. Click on the header of any column of the table. 2. Click on Columns in the context menu. 3. Remove the check mark at the name of the column that you want to hide by clicking on it.	Selected column is no longer shown.
Test unhide a metric	1. Click on the header of any column of the table. 2. Click on Columns in the context menu. 3. Add a check mark at the name of the column that you want to unhide by clicking on it.	Selected column is shown in the table.

Test Show trend view

Table 7.7 documents the system test cases performed to test the SF Show trend view. Due to time constraints this system function was implemented only on client-side, i.e. it could be solely tested with the mock server.

Table 7.7: System tests for SF Show trend view.

Test case	Test steps	Actual result
Test show trend view for number of LOC	1. Click on any file name of the table. 2. Click on Trend LOC tab.	Trend chart for the number of LOC of the selected file is shown.
Test show trend view for unit test coverage	1. Click on any file name of the table. 2. Click on Trend Coverage tab.	Trend chart for the unit test coverage of the selected file is shown.

Test Save selected filters

Table 7.8 documents the system test case performed to test the SF Save selected filters. The precondition for this test is to filter the files beforehand, either by Component Group name or Custom Code Group name. This system test was conducted for both filters.

Table 7.8: System test for SF Save selected filters.

Test case	Test steps	Actual result
Test save selected filters	Save the URL, which new contains the filter criteria as query parameters as a bookmark in your browser.	Bookmarked URL is added to your browser.

Test Apply saved filters

Table 7.9 documents the system test case performed to test the SF Apply saved filters. The system precondition for this test is to have a custom filter (Component Group filter or Custom Code Group filter) already saved as a bookmark to your browser. This system test was conducted for both filters.

Table 7.9: System test for SF Apply saved filters.

Test case	Test steps	Actual result
Test apply saved filters	Open the bookmarked URL in your browser.	Quality Page displays files which meet filter criteria saved as URL parameters.

7.3.3 NFR: Performance efficiency (Time-behavior)

To assess the fulfillment of the NFR formulated in Table 5.12, we used the Network Panel of the Google Chrome browser. The Network panel shows the list of all retrieved resources with their respective loading times. We performed multiple measurements, all of which did not exceed five seconds (3-4 seconds on average), except for those when the network or the database were unavailable.

8 Evaluation

This chapter presents the results of the feedback meeting conducted in the last phase of this project. Previously interviewed software developers were invited. This meeting served as the basis for the evaluation of the Quality Page.

8.1 Feedback Meeting

The software developers interviewed in the first phase of this work were invited to a feedback meeting, at which six out of the eight interviewees participated. We decided to conduct a group meeting to initiate a broader discussion than can be achieved in a one-on-one conversation.

8.1.1 Method

At the feedback meeting the Quality Page in its current implementation state was presented. Both views – *Table* and *Trend* – were presented and the respective system functions were called. There was a pause after each user input for questions and suggestions.

The presentation was performed with mock data, therefore the response time of the UI could not be evaluated during this meeting.

8.1.2 Results

The UI of the prototype was considered as easy to understand and the Quality Page in general was perceived as helpful. One developer pointed out that compared to the static analysis tool he currently uses, this application provides a more focused view on code quality as it does not “overspam” the user with warnings.

Following suggestions were made:

Trend View

One of the participants said that the view may be too fine granular and that the metric should not be calculated and shown for each day but only for the days when the file was changed. On the other hand, another interviewee pointed out, that when you have just changed a file, you would want to see your progress immediately. In this case, a daily trend would make sense.

One developer pointed out that it would be helpful to have the possibility to see the commit hash to be able to check in the VCS which commit is responsible for a change in the graph.

Table View

One participant pointed out that it would be helpful to be able to jump from the file name directly to the code browser to view the relevant part of code.

Another participant pointed out that this is unrealistic that somebody will use Custom Code Groups to manage their test files. It would be much practical to have an option to filter out all test file by e.g. a checkbox “Include test code”. Yet another interviewee said that updating Custom Code Groups is not something you do very often.

Some of the participants needed clarification about the connection between file-level and method-level metrics. Implementing method-level metrics would have gone beyond the scope of this work and therefore, they are not yet supported by the Quality Page. The exact question was “When the number of cyclomatic complexity will also be included in the analysis, how will it be displayed in the Table View? Also for a file?”.

General suggestions

Some of the developers expressed the need to be able to declare custom metrics which better fit team-specific quality needs. As an example, the participant named TODO tags in comments.

8.2 Discussion and Future Work

Our general impression was that the software developers would be willing to try using this tool in their daily work and perceived the visual representation of metrics as helpful.

An important insight was the realization, that the software developers do not consider Custom Code Groups as flexible as we have assumed. Grouping files with this method is not something that a developer is willing to do every day. We have to keep this in mind for future extensions of filtering functionality. This is important to be able to provide a solution that is practical and does not impose additional effort which is not well-founded in the eyes of practitioners.

Based on the results of this feedback meeting, we would propose following directions to improve and to extend the functionalities of the Quality Page:

- Enabling integration of the Quality Page with the online code browser
- Extending the filtering for test code (independent from Custom Code Groups, with string matching, for instance)
- Enriching the trend chart with the information on the commit
- Enabling configurable granularity of the time axis on the trend chart
- Introducing method-level metrics
- Enabling the user to define own custom or team-specific metrics

9 Conclusion

This chapter presents the main results of this work and describes ideas for the future.

9.1 Summary and Discussion

The overall goal of this work was to design and implement a web application (Quality Page) which would display code quality metrics and motivate the users – software developers – to improve their code. To achieve this goal, we first investigated the problem by conducting two studies: a systematic study of the relevant literature and an empirical case study for which we interviewed potential users of the application – software engineers at SAP HANA. With both studies, we intended to find answers to the following research questions:

Which metrics do software developers need?

How can metrics be transferred into concrete tasks?

How can metrics be visualized?

In our literature study we examined four different approaches to actionable code analysis and metric-based quality assurance. One of the main insights from this study was the fact that developers should be able to customize their analysis tools, i.e. decide themselves which metrics they want to see. Additionally, displaying a threshold for a metric was discovered as an important element in ensuring actionability of a metric. And the concept of a customizable application would be even better supported with a configurable threshold.

The results of the interview study confirmed the need for customization of the tool. However, the main result of the interview study was a list of metrics (provided in 4.2.2) developers consider useful. Another conclusion drawn from this study was the fact that there should be a possibility for a software developer to exclude certain files from the analysis and that more sophisticated filtering possibilities than just assigning of a file to a certain HANA component would be helpful.

From the insights gained from both studies, requirements for the Quality Page were

derived. These requirements served as the basis for the design and subsequent implementation of the tool prototype. The Quality Page in its current implementation state supports the visualization of two metrics (number of LOC and unit test coverage). For each metric, the user can view the current value or a trend over time. Multiple options including filtering enable a customized view of the metrics. The actionability of the metrics is supported with an indicated threshold and possibility to sort the files by their metric value to quickly detect problematic parts of code. Due to time restraints, not all requirements could be realized as specified. E.g. instead of a configurable threshold for a metric, a threshold in both *Table* and *Trend* views are indicated based on a fixed value defined in code.

The evaluation carried out together with the participants of the interview study confirmed that we are on the right track with the Quality Page but the end has not been reached yet and there are several directions that can be pursued to improve and extend the Quality Page.

9.2 Outlook

Due to the extensible application structure, new metrics from the aforementioned list can be added to the frontend with low development effort. As to the backend application, for some metrics on the list, the object-oriented metrics, for instance, the process of metric data generation and collection has first to be established.

A meaningful visualization of such metrics as cyclomatic complexity requires a drill-down from the file level to the method level, maybe with a halt in between for the class level. Designing and implementing such drill-down would certainly be a logical extension of the current implementation.

Creating a closer connection between the metric and the code would improve the aspect of actionability as the user could directly jump into exploring the code. This could be accomplished through integrating already existing online code browser with the Quality Page.

Making the threshold for metrics and the time frame for the trend chart configurable certainly represent welcomed steps toward customization. Furthermore, enabling the user to define his or her own metrics would be a goal worth aspiring to in the long run.

10 Bibliography

- [1] S. Akbarinasaji, B. Soltanifar, B. Çağlayan, A. B. Bener, A. Miransky, A. Filiz, B. M. Kramer, & A. Tosun, “A metric suite proposal for logical dependency”, in *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, ser. WETSoM '16, Austin, Texas: ACM, 2016, pp. 57–63. DOI: [10.1145/2897695.2897704](https://doi.org/10.1145/2897695.2897704).
- [2] A. Bergel, S. Denier, S. Ducasse, J. Laval, F. Bellingard, P. Vaillergues, F. Balmas, & K. Mordal-Manet, “Squale – software quality enhancement”, in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 285–288. DOI: [10.1109/CSMR.2009.13](https://doi.org/10.1109/CSMR.2009.13).
- [3] P. Bourque & R. E. Fairley, “Guide to the software engineering body of knowledge (swebok(r)): Version 3.0”, 2014.
- [4] S. R. Chidamber & C. F. Kemerer, “A metrics suite for object oriented design”, *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994. DOI: [10.1109/32.295895](https://doi.org/10.1109/32.295895).
- [5] M. Christakis & C. Bird, “What developers want and need from program analysis: An empirical study”, in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 332–343. DOI: [10.1145/2970276.2970347](https://doi.org/10.1145/2970276.2970347).
- [6] “Ieee standard for a software quality metrics methodology”, *IEEE Std 1061-1998*, pp. i–, 1998. DOI: [10.1109/IEEESTD.1998.243394](https://doi.org/10.1109/IEEESTD.1998.243394).
- [7] S. H. Kan, *Metrics and models in software quality engineering*, eng, 2. ed. Boston: Addison-Wesley, 2002, Online–Ressource (xxvii, 528 p.) Mode of access: World Wide Web. - Includes bibliographical references and index. - Made available through: Safari Books Online, LLC.
- [8] B. A. Kitchenham & S. Charters, *Guidelines for performing Systematic Literature Reviews in Software Engineering (Version 2.3)*. 2007.
- [9] M. Kuuttila, M. Mäntylä, U. Farooq, & M. Claes, “Time pressure in software engineering: A systematic literature review”, *CoRR*, vol. abs/1901.05771, 2019. arXiv: [1901.05771](https://arxiv.org/abs/1901.05771).

- [10] S. Martínez-Fernández, P. Jovanovic, X. Franch, & A. Jedlitschka, “Towards automated data integration in software analytics”, in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, ser. BIRTE ’18, Rio de Janeiro, Brazil: ACM, 2018, 6:1–6:5. DOI: [10.1145/3242153.3242159](https://doi.org/10.1145/3242153.3242159).
- [11] S. Martínez-Fernández, A. Jedlitschka, L. Guzmán, & A. M. Vollmer, “A quality model for actionable analytics in rapid software development”, in *2018 44th Euro-micro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 370–377. DOI: [10.1109/SEAA.2018.00067](https://doi.org/10.1109/SEAA.2018.00067).
- [12] K. Mordal-Manet, J. Laval, S. Ducasse, N. Anquetil, F. Balmas, F. Bellingard, L. Bouhier, P. Vaillergues, & T. J. McCabe, “An empirical model for continuous and weighted metric aggregation”, in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 141–150. DOI: [10.1109/CSMR.2011.20](https://doi.org/10.1109/CSMR.2011.20).
- [13] D. Nicolette, *Software development metrics*, eng, Online-Ausg. Shelter Island, NY: Manning, 2015, Online–Ressource.
- [14] M. Orru & M. Marchesi, “A case study on the relationship between code ownership and refactoring activities in a java software system”, in *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, ser. WETSOM ’16, Austin, Texas: ACM, 2016, pp. 43–49. DOI: [10.1145/2897695.2897702](https://doi.org/10.1145/2897695.2897702).
- [15] B. Paech & A. Kleebaum, *Richtlinien zur literaturrecherche*, 2016.
- [16] B. Paech & K. Kohler, *Task-driven requirements in object-oriented development*, 2004. DOI: [10.1007/978-1-4615-0465-8_3](https://doi.org/10.1007/978-1-4615-0465-8_3).
- [17] “Proceedings of the 7th international workshop on emerging trends in software metrics”, eng, Association for Computing Machinery-Digital Library & ACM Special Interest Group on Software Engineering, New York, NY: ACM, 2016, 1 online resource (76 pages).
- [18] P. Rachow, S. Schröder, & M. Riebisch, “Missing clean code acceptance and support in practice - an empirical study”, in *2018 25th Australasian Software Engineering Conference (ASWEC)*, 2018, pp. 131–140. DOI: [10.1109/ASWEC.2018.00026](https://doi.org/10.1109/ASWEC.2018.00026).
- [19] P. Runeson, *Case study research in software engineering, Guidelines and examples*, eng. Hoboken, NJ: Wiley, 2012, xviii, 237 p.
- [20] C. Sadowski, J. v. Gogh, C. Jaspan, E. Söderberg, & C. Winter, “Tricorder: building a program analysis ecosystem”, in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 598–608. DOI: [10.1109/ICSE.2015.76](https://doi.org/10.1109/ICSE.2015.76).

- [21] S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit, & A. Trendowicz, “Operationalised product quality models and assessment: The quamoco approach”, vol. 62, 2015, pp. 101–123. DOI: <https://doi.org/10.1016/j.infsof.2015.02.009>.
- [22] R. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014. DOI: [10.1007/978-3-662-43839-8](https://doi.org/10.1007/978-3-662-43839-8).

List of Figures

3.1	The dashboard of Q-Rapids tool.	24
3.2	Quamoco tool: Sunburst visualization of quality factors.	25
3.3	Quamoco tool: Kiviat diagram comparing two programs.	26
3.4	Example of Tricorder’s robocomment.	27
5.1	Domain Data Diagram.	49
5.2	User Interface Structure Diagram.	51
6.1	Deployment Diagram of QualityPage.	53
6.2	Class Diagram of the Frontend Application.	55
6.3	Implementation of SF Show current metric value.	56
6.4	Class Diagram for the implementation of SF Filter files.	57
6.5	Implementation of SF Filter files for Custom Code Groups.	58
6.6	Implementation of SF Filter files for Component Groups.	58
6.7	Class Diagram for the implementation of SF Sort files.	59
6.8	Implementation of SF Sort files.	59
6.9	Class Diagram for the implementation of SF Search for Files.	60
6.10	Decision knowledge for the Search for Files implementation.	61
6.11	Implementation of SF Search for files.	61
6.12	Class Diagram for the implementation of SF Hide / unhide a metric.	61
6.13	Implementation of SF Hide / unhide a metric.	62
6.14	Class Diagram for the implementation of SF Show trend view.	62
6.15	Implementation of SF Show trend view.	63

List of Tables

2.1	Overview of code-relevant metrics.	9
3.1	Research questions derived from the project's goal.	14
3.2	Criteria for selection of relevant articles.	16
3.3	Results of the snowballing method.	16
3.4	Overview of found articles.	17
3.5	Comparison of the Q-Rapids, Squale, Quamoco and Tricorder approaches.	19
4.1	Overview of the answers to the interview questions on the usefulness of metrics.	36
5.1	Persona: Software developer.	42
5.2	Show current metric value.	44
5.3	Filter files.	45
5.4	Sort files.	45
5.5	Search files.	46
5.6	Hide/unhide a metric.	46
5.7	Show trend view.	47
5.8	Create threshold for a metric.	47
5.9	Update threshold for a metric.	47
5.10	Save selected filters.	48
5.11	Apply saved filters.	48
5.12	NFR: Performance efficiency (Time-behavior)	50
5.13	NFR: Maintainability (Reusability)	50
5.14	Overview of functional and non-functional requirements.	51
7.1	Number of LOC per file.	66
7.2	System test for SF Show current metric value.	68
7.3	System tests for SF Filter files.	69
7.4	System tests for SF Sort files.	69
7.5	System tests for SF Search for files.	70
7.6	System tests for SF Hide / unhide a metric.	70

7.7	System tests for SF Show trend view.	71
7.8	System test for SF Save selected filters.	71
7.9	System test for SF Apply saved filters.	71

Glossary

actionable Ability of an information to be directly transferable into a concrete action or a task. 14, 23, 78

controls UI elements which can be used independently, e.g. Button, Table etc. 56

diff Type of test coverage. Diff coverage is test coverage for the newly submitted code only. 9, 35

event handler Function dedicated to reacting to a specific user action, such as clicking on a button or pressing a key. 64

full Type of test coverage. Full coverage includes coverage achieved through unit tests, integration tests and end-to-end tests. 9

metric Value resulted from measuring a certain attribute of a program. 4

unit Type of test coverage. Unit coverage is test coverage achieved through unit tests. 9

Acronyms

HTTP Hypertext transfer protocol. 54

JSON JavaScript Object Notation. 54, 57

LOC Lines of Code. 4, 8, 41, 57, 68, 84

REST Representational state transfer. 54

SF System function. 43, 52, 70–73, 84, 85

ST Subtask. 43

UI User Interface. 58, 73

URL Uniform resource locator. 44, 54, 73

UT User task. 43

VCS version control system. 12, 68, 69, 76

WS Work space. 44, 64

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht.

Die Arbeit ist in gleicher oder vergleichbarer Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Heidelberg, den