



Utilizing Code Coverage Density to Enhance Software Quality Management Decisions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Patrick Fleck

Matrikelnummer 01025484

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Thomas Grechenig

Wien, 10. Mai 2022

Unterschrift Verfasser

Unterschrift Betreuung



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Utilizing Code Coverage Density to Enhance Software Quality Management Decisions

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Patrick Fleck

Registration Number 01025484

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 10. Mai 2022

Signature Author

Signature Advisor



Utilizing Code Coverage Density to Enhance Software Quality Management Decisions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Patrick Fleck

Matrikelnummer 01025484

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

Betreuung: Thomas Grechenig

Wien, 10. Mai 2022



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Patrick Fleck

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

Wien, 10. Mai 2022

Patrick Fleck



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich möchte mich bei meiner Familie und meinen Freunden für ihre Unterstützung während meines gesamten Studiums bedanken. Ganz speziell möchte ich mich bei meiner Ehefrau bedanken, da sie während meiner Studienzeit und der Durchführung dieser Diplomarbeit sehr viel Geduld für mich aufbringen musste.

Außerdem möchte ich mich bei meinem Arbeitgeber und meinen Arbeitskolleg/innen bedanken, welche mir einen idealen Rahmen für die Kombination aus Studium und Berufstätigkeit schafften.

Zu guter Letzt möchte ich ganz besonders die Unterstützung meiner Betreuer im Zuge dieser Diplomarbeit hervorheben, ohne die ein erfolgreicher Abschluss der Arbeit nicht möglich gewesen wäre.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I want to thank my family and friends for their support and encouragements that carried me through my studies. A special acknowledgement goes to my wife for having great patience with me while conducting my studies and this master's thesis.

Furthermore, I want to thank my employer and work colleagues that enabled a reasonable balance between my professional activities and my studies.

Finally, I want to acknowledge the support and encouragement of my advisors, which was an important factor for the success of this thesis.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Aus heutiger Sicht sind Tätigkeiten und Entscheidungsprozesse in den Bereichen Software-Qualitätsmanagement und -Qualitätssicherung zu essenziellen Bestandteilen gängiger Software-Entwicklungsprozesse avanciert. Es existieren vielfältige Software-Metriken, welche im Bereich der Software-Entwicklung tätigen Personen als Grundlage für Entscheidungen im Rahmen von Maßnahmen zur Verbesserung und Aufrechterhaltung der Qualität eines Software-Produkts dienen.

Obwohl erwähnte Metriken bereits sehr gut in bestehende praxisorientierte Werkzeuge (z.B.: Code Coverage Tools) integriert wurden, zeigt diese Diplomarbeit, dass dennoch zahlreiche schwer erfüllbare Informationsbedürfnisse im Bereich Software-Qualitätsmanagement und -Qualitätssicherung existieren. Im Speziellen werden im Zuge dieser Arbeit prominente Informationsbedürfnisse in diesem Kontext gesammelt, diskutiert und aktuell verfügbaren Code Coverage Tools gegenübergestellt, welche auf deren Erfüllung abzielen. Diese Recherche in Bezug auf den aktuellen Stand der Technik zeigt, dass derzeit eklatante Informationslücken in den Bereichen *Risikoanalyse von Codeänderungen und Refactorings*, *Wartung von Test Suites* und der *Analyse von Testredundanzen* bestehen.

Auf dieser Basis wird in dieser Diplomarbeit die Metrik *Code Coverage Density* eingeführt, welche klassische Code Coverage um die Beschreibung der Verteilung der Testfälle über ein bestimmtes Software-Artefakt erweitert. Des Weiteren umfasst diese Arbeit die Implementierung eines Prototyps, welcher auf dieser Metrik basierende Features bzw. Visualisierungen zur Verfügung stellt und auf die Erfüllung der identifizierten Informationsbedürfnisse abzielt. Zudem wird eine szenariobasierte Expertenevaluierung unter Zuhilfenahme eines vom Prototyp generierten Coverage Density Reports durchgeführt. Die Ergebnisse dieser Evaluierung zeigen, dass die eingeführte Metrik und die darauf aufbauenden Visualisierungen ein bemerkenswertes Potenzial zur Unterstützung bei Tätigkeiten in den Bereichen *Risikoanalyse von Codeänderungen und Refactorings* und *Wartung von Test Suites* mit sich bringen. Es wird außerdem gezeigt, dass das eingeführte Konzept im Rahmen der *Analyse von Testredundanzen* bei detaillierteren Untersuchungen von vorab als potentielle Redundanzen identifizierten Tests beitragen kann, jedoch nicht zur Auffindung von ebendiesen geeignet ist.

Keywords: *Software-Qualitätsmanagement, Software-Qualitätssicherung, Software-Testen, Code Coverage, Coverage-Metriken, Coverage-Visualisierung, Software-Wartung*



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

From today's perspective, activities and decision-making processes in software quality management and assurance have become an inevitable element of current software engineering processes. There exists a variety of software metrics that serve software engineers as a decision basis during actions that aim at enhancing and maintaining a software product's quality.

This master's thesis shows that though such metrics have been gracefully transferred into practical tools (i.e., code coverage tools), there are information needs in the context of software quality management and assurance that are still hard to satisfy. In fact, popular unsatisfied information needs are gathered, discussed and reflected concerning scientific approaches and currently available code coverage tools that try to satisfy them. This research on the state of the art shows that there are significant information gaps in the areas of *risk assessment concerning code changes and refactorings*, *test suite maintenance* and *test redundancy analysis*.

On this basis, this thesis introduces the novel software metric *code coverage density*, which enhances classical code coverage by the notion of describing how test cases distribute over a software product's artifacts. Furthermore, the thesis includes the implementation of a prototype that utilizes the latter and provides features and visualizations that are concentrated on fulfilling the found unsatisfied information needs. Moreover, a scenario-based expert evaluation is conducted with the aid of a coverage density report generated by the implemented prototype. The results of the latter show that the introduced metric along with the proposed visualizations has a significant potential for filling the identified information gaps in the areas of *risk assessment concerning code changes and refactorings* and *test suite maintenance*. Concerning the field of *test redundancy analysis*, it is shown that the presented concept rather establishes assistance for further analysis of potential test redundancy suspects than providing support for discovering those.

Keywords: *Software Quality Management, Software Quality Assurance, Software Testing, Code Coverage, Coverage Metrics, Coverage Visualization, Software Maintenance*



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	xiii
Abstract	xv
Contents	xvii
1 Introduction	1
1.1 Problem Description and Motivation	1
1.2 Expected Results	3
1.3 Methodology	3
1.4 Contributions	5
1.5 Structure	5
2 Fundamentals	7
2.1 Software Quality Assurance	7
2.2 Software Metrics	10
2.3 Software Testing	11
2.4 Refactoring	16
3 State of the Art	19
3.1 Related Work	19
3.2 State of the Art Tools	24
3.3 Summary	32
4 Metric Definition	35
4.1 Line Coverage Density	35
4.2 Method Coverage Density	36
4.3 Class Coverage Density	37
4.4 Higher Granularity Levels	38
5 Concept	39
5.1 Concept Proposal	39
5.2 Expert Interviews	47
5.3 Adaptions	55
	xvii

5.4	Final Requirements	57
6	Implementation	59
6.1	Execution Phases	59
6.2	Technical Concept	61
6.3	Results	65
7	Evaluation	73
7.1	Fundamental Considerations	73
7.2	Example Project	74
7.3	Procedure	75
7.4	Participants	76
7.5	Scenarios	76
7.6	Results	85
7.7	Discussion	96
8	Conclusion	101
8.1	Future Work	102
A	Appendix	105
A.1	Expert Interview Questionnaires	105
A.2	Evaluation Interview Questionnaires	118
A.3	Listings	124
	List of Figures	127
	List of Tables	131
	Listings	131
	Acronyms	133
	References	135
	Online References	139

Introduction

Software quality management is a crucial discipline of today's software engineering processes. Engineers make use of diverse metrics in this context to establish statements about software artifacts and make decisions based on them. Code coverage is a metric that states the percentage of implemented code which is covered by test cases [1]. There are numerous tools available that, on the one hand, establish the metric itself for a given test run and, on the other hand, generate high sophisticated test reports which support decision-making. Some of those tools show how often a Line of Code (LOC) is covered by tests. In blogs and forums [51], [52], the aggregated line coverage frequency of multiple test cases for a given source code has been referred to as *code coverage density*. However, this term is neither common in the field of software quality management, nor officially standardized or scientifically acknowledged.

1.1 Problem Description and Motivation

Prior to this master thesis, research on currently available code coverage tools was made. The target was to find out if there are tools that already implement computations and visualizations following the notion of *code coverage density*. The observations showed the following:

- Only half of the analyzed tools visualize how often a piece of code was covered (see Section 3.2).
- The visualizations concerning how often a part of the code was covered is only given on the lowest level (i.e., on line level), but not in broader scopes (i.e., on class or package level).
- The color visualizations for covered code do only state whether a line has been covered or not, e.g., by highlighting covered lines green and uncovered ones red.

- Though there are visualizations that show which test cases run through which LOCs (and also higher-level artifacts, see Section 3.2), the analyzed tools do not put enough emphasis on imparting a clear understanding between source code and test cases.
- In analogy to the latter, the tools do not provide enough support for finding and analyzing test case duplicates.

From a practical point of view, there are certain scenarios that would benefit from focusing more towards the idea and analysis of *code coverage density*:

- **Impact of code changes and refactorings:** Developers often fear these actions, as they contain a certain risk that parts of the program may be altered in a non-expected way. Unit test coverage often relieves the worries about a planned refactoring because it clearly shows which parts of the code are tested and which are not. However, 100% unit test coverage may not be sufficient if we assume that certain parts of a program or application may have functionality in an integrated or system-scoped way (e.g., communicate with other program units, components or services). Therefore, integration and system testing is needed to additionally cover this behavior. With today's tools and visualizations, it may be very hard to state which and how many tests run through a given program code and if a piece of code is tested considering multiple scopes. *Code coverage density* along with further analysis and visualization of this metric could give more insight in this area. Namely, it could give better insight on how risky a code change or refactoring may be.
- **Critical program parts:** The prior also leads to the idea that *code coverage density* could state to which extent critical parts of the code are tested in a proper way, especially when thinking about the integration of the program components and their mutual interactions with each other. Analyzing and visualizing *code coverage density* could state that e.g., integration tests for a certain part of the application may be missing.
- **Test suite maintenance:** The idea of *code coverage density* could have a positive impact on test suite maintenance, e.g., when trying to find test duplicates. Based on current tool support, such duplicates can only be found by searching for tests that call a certain part of the program and analyzing it. If it is clearly stated which tests run through which artifact, it would be much easier to identify them, as well as deciding if there are test duplicates or not.

1.2 Expected Results

At a glance, the main objective of this thesis is to explore how the novel metric *code coverage density* helps software engineers in their daily work processes. In detail, the investigations focus on how the metric could support in the following areas:

- *Risk assessment of code changes and refactorings*
- *Code coverage and coverage distribution assessment*, i.e., the identification of critical program part coverage, with reference to coverage scope (e.g., integration tests, system tests)
- *Identification of test redundancies and overlaps*

The derived research questions that are pursued and answered during this thesis are, therefore, as follows:

RQ 1: How does *code coverage density* support software engineers in deciding how risky a code change or refactoring is?

RQ 2: How does *code coverage density* support software engineers in assessing the coverage distribution of a project?

RQ 3: How does *code coverage density* support software engineers in identifying test redundancy and overlaps?

Note that the term “software engineers” includes software developers, testers, as well as quality managers.

1.3 Methodology

In the following, the methodological approaches that are employed during this thesis are depicted in detail. In general, the overall procedure of the thesis is divided into four phases, which are depicted in Figure 1.1 and described in detail in the following subsections.

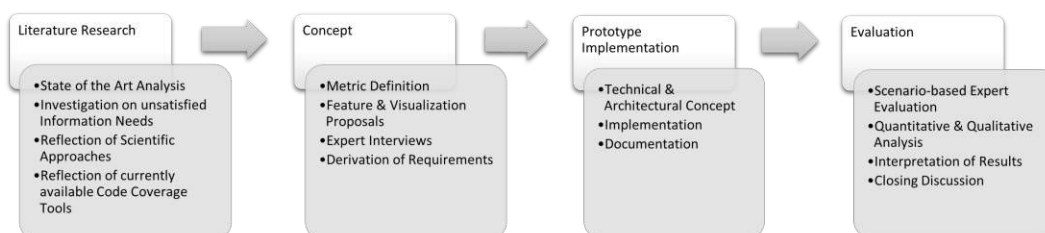


Figure 1.1: Methodology outline

1.3.1 Literature Research

Phase 1 consists of a scientific research in the field of information needs in software engineering. The target is to identify and summarize important findings of scientific surveys and investigations in the context of hardly satisfiable information needs that are in touch with the initial ideas and notions of this thesis.

As a second step, further research concerning scientific approaches that attempt to satisfy the identified information needs is conducted. Moreover, currently available code coverage tools are evaluated with a focus on the question if, on the one hand, they offer relevant features for fulfilling the identified information needs, and on the other hand, the encountered scientific approaches have been transferred into practice-oriented tools.

The outcome of this phase is a reflection and interpretation of those findings and states which information needs are still not fulfilled by scientific approaches and code coverage tools. This furthermore emphasizes the relevance of this thesis and especially the defined research questions.

1.3.2 Concept Evaluation

The second phase concentrates on the question how the unfulfilled information needs gathered in phase 1 could be satisfied by the novel metric *code coverage density* and possible visualizations that build upon it. For this target, the first step is to establish a complete and sound definition of the metric. Secondly, requirements for a prototype of a potential novel code coverage analysis tool are defined, which aims at fulfilling the found information needs with the help of the defined metric. Moreover, concrete realization proposals in the form of visualization drafts and mockups are established.

On this basis, interviews with selected experts are conducted. The intention of those is, on the one hand, to verify the completeness, soundness and understandability of the metric itself and, on the other hand, to present and evaluate the visualization and feature proposals, as well as ranking them according to their relevance. Moreover, the target is to gather further ideas and information needs in this context and to validate the idea and notion of this thesis. The outcome is a final list of ranked requirements for the planned prototype, which form the basis for phase 3.

1.3.3 Prototype Implementation

In phase 3, a concrete prototype, which generates a static coverage density report, is implemented based on the concepts and requirements established in phase 2. The process follows an iterative approach whereby after each iteration, the currently available version of the prototype is examined and encountered problems/difficulties, as well as potential adoptions and extensions of the initial requirements are documented. Furthermore, the process is documented with regard to the technical and architectural concept.

The final outcome is a working prototype, which generates a static analysis report satisfying the requirements defined phase 2.

1.3.4 Evaluation

In phase 4, a scenario-based expert evaluation is conducted with the aid of a coverage density report generated by the established prototype for a provided software project. The participants are solving pre-defined scenarios and problems in this context. Moreover, the process is supported by both qualitative and quantitative questions, which aim at investigating on the degree of support that the report (and therefore also the defined metric) gives while acting out the scenarios. Moreover, general discussions on further remarks and possible improvements are conducted.

The outcomes are afterwards analyzed, evaluated and finally interpreted in order to answer the defined research questions in Section 1.2. As a final step, the results retrieved through the thesis are reflected and summarized.

1.4 Contributions

Firstly, this thesis establishes a confrontation of unsatisfied information needs in software engineering with scientific approaches and features in currently available code coverage and testing tools. Secondly, fundamental research on the notion of *code coverage density* is conducted by showing how an extension of coverage data by the means of this notion could fill the information gaps encountered and help engineers with their daily work. In the big picture, the thesis shall also motivate further research in this field by building awareness that enhanced processing and visualization of coverage data, in particular concerning the notion of *code coverage density*, is sensible and may support decision-making in various areas.

1.5 Structure

The rest of this thesis is structured as follows.

Chapter 2 imparts basic fundamental definitions needed in the context of this thesis. In Chapter 3, the findings and results of the conducted scientific research are presented. A definition of the novel metric *code coverage density* is given in Chapter 4. Further outcomes of the concept evaluation are discussed in Chapter 5. Chapter 6 depicts the architectural and technical concept for the implementation phase and furthermore the actual implemented prototype. The evaluation phase is handled in Chapter 7. Finally, a summary of the findings and a discussion on potential future work is presented in Chapter 8.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Fundamentals

In this chapter, fundamental definitions and principles that are relevant for this thesis are given. The main areas that are covered are the principles and basics of software quality assurance and management (Section 2.1), software metrics (Section 2.2) and software testing (Section 2.3). Finally, Section 2.4 imparts basic knowledge in the area of refactoring activities.

2.1 Software Quality Assurance

The upcoming sections present the fundamental definitions for the terms *software quality*, *software quality assurance* and *software quality management*.

2.1.1 Definition of Software Quality

The official standard ISO 9000 [2] specifies the basic term *quality* as follows:

“Quality: degree to which a set of inherent characteristics of an object fulfills specified quality requirements.”

In the context of software products, organizations such as the Institute of Electrical and Electronic Engineers (IEEE) and the International Organization for Standardization (ISO) established corresponding standards that form the basis for software quality assessment. The most recent one is the standard ISO/IEC 25010 [3], which defines *software quality* more specifically as follows:

“The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value.”

The stated needs are furthermore represented by defined *quality models* that categorize product quality into well-defined characteristics and subcharacteristics. At a glance, the standard states the following characteristics groups (Figure 2.1) [3]:

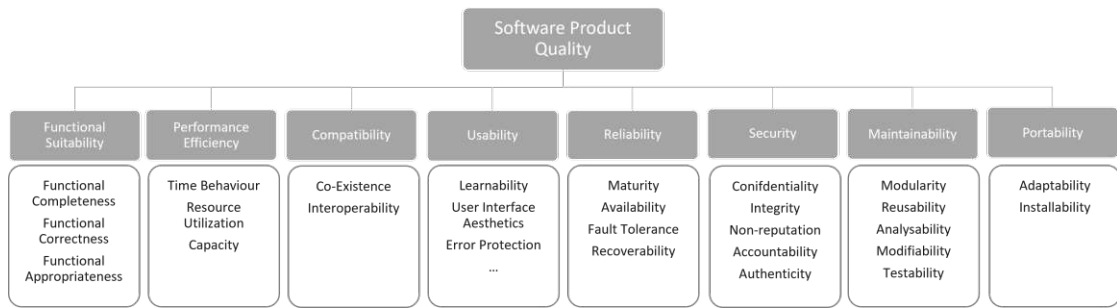


Figure 2.1: Software Product Quality [3]

- **Functional suitability** describes to which degree the product provides functionality that suffices stated and implied needs when used under specified conditions. The term is furthermore divided into the more detailed subcharacteristics *functional completeness*, *correctness* and *appropriateness*.
- **Performance efficiency** describes a group of characteristics that state the performance relative to the amount of resources used under certain conditions, i.e., *time behavior*, *resource utilization* and *capacity*.
- **Compatibility** is a characteristic defined within two different scopes. *Co-existence* describes the degree to which a product is able to perform its functions efficiently while sharing its environment and resources with other products. *Interoperability* describes the degree to which two or more systems, products or components can exchange information and process the information that has been exchanged.
- **Usability** is defined as the degree to which the product can be used by users in order to reach specified goals with a certain level of effectiveness, efficiency and satisfaction. Usability is furthermore divided into more detailed sub-characteristics such as *learnability*, *user interface aesthetics* or *user error protection*.
- **Reliability** is the degree to which a product performs particular functions under specified conditions within a defined period of time and is further divided into the characteristics *maturity*, *availability*, *fault tolerance* and *recoverability*.
- **Security** is described as the degree to which a product protects information and data so that other entities only have access to data that their types and levels of authorization correspond to.
- **Maintainability** is defined as the degree of effectiveness and efficiency with which a product can be maintained, which moreover includes the characteristics *modularity*, *reusability*, *analyzability*, *modifiability* and *testability*.

- **Portability** is, in analogy to maintainability, the degree of effectiveness and efficiency with which a product can be transferred from one environment (e.g., hardware, software, etc.) to another and is moreover classified with the corresponding terms *adaptability* and *installability*.

2.1.2 Software Quality Assurance

The IEEE defines Software Quality Assurance (SQA) as a set of activities that define and assess the adequacy of software processes in order to provide evidence that the processes are appropriate and produce software products with suitable quality for their intended purposes [4]. Common literature usually simplifies this definition and states that SQA is the collectivity of all activities that aim at ensuring a proper quality of a software product (i.e., with respect to defined quality requirements as described in 2.1.1) [5]–[7]. Those activities can furthermore be divided into *constructive* and *analytical* methods:

- **Constructive SQA** includes all quality ensuring activities during the development of software. This includes i.a. the agreement on specific methods, tools, design principles and process models that ensure that the product can achieve a certain level of quality [5], [7].
- **Analytical SQA** checks if the quality requirements are fulfilled by the product and tries to identify faults and deficits that impede the correspondence to those [5]–[8]. The methods employed are moreover distinguished as **static** and **dynamic analytical methods**. Static methods do not execute the object to analyze and consist of activities like *static code analysis*, *code reviews*, *walkthroughs* and *inspections*. In contrast, dynamic methods analyze and monitor a products behavior with respect to the specified behavior. Activities conducted in this area are i.e., *software testing* and *dynamical analysis* (i.e., assessing the run time behavior of the software with reference to performance or resource usage) [5], [7], [8].

2.1.3 Software Quality Management

Software Quality Management (SQM) is a discipline which is situated on top of SQA. The term describes a collection of all processes that ensure that software products, services and life cycle process implementations meet an organizations software quality objectives and lead to stakeholder satisfaction [9]. In addition to SQA, SQM includes the following three subcategories [9]:

- Software Quality Planning (SQP), which includes deciding which quality standards to target, defining quality goals and estimating the effort and schedule of software quality ensuring activities
- Software Quality Control (SQC), which includes the analysis and evaluation of specific project artifacts in order to determine whether they comply with the established quality standards (i.e., requirements, designs, contracts and plans)

- Software Process Improvement (SPI), which includes improving the software process effectiveness, efficiency and other characteristics with a special target on improving the software quality

2.2 Software Metrics

Based on the standard IEEE 1061 [10], a *software metric* is defined as follows:

“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.”

Such a metric is therefore measurable in a quantitative way and allows the interpretation of software quality. In general, there is a variety of software metrics which are usually classified either as *process metrics* or *product metrics* [1], [8]. Note that in the context of this thesis, the latter are of higher importance and therefore described in more detail in the following.

Process metrics deal with data of the software engineering process in general and therefore describe the characteristics of the process that leads to a software product [8], [10], [11].

Product metrics measure the artifacts and deliverables that result from a process activity without respect to the latter [8], [10], [11]. In contrast to process metrics, product metrics are specifically used for analytical SQA activities which is of special interest in the context of this thesis.

In the context of static analytical SQA, there exist a variety of *code metrics* that can be established without executing the object to analyze. Examples for such metrics are i.a.:

- **Size metrics**, such as LOC or Number of Statements (NOS), which are employed to estimate and limit the size of artifacts in order to, e.g., calculate prices for a product [8]
- **Object oriented metrics**, such as Coupling between Object (CBO), Depth of Inheritance Tree (DIT) or Lack of Cohesion in Methods (LCOM), which describe the relations between objects within the software [1], [8]
- **Complexity metrics**, such as function point metrics, the Halstead metric or the McCabe metric [8], [12]

In contrast, there also exist numerous dynamic product metrics which require that the product itself is available in an executable state for dynamic analytical SQA purposes. Examples for such metrics are i.a.:

- **Mean Time to Failure (MTTF)**, which measures the time between two consecutive failures [13]
- **Defect Density**, which describes the number of defects relative to a products size (i.e., with respect to formerly described size metrics) [13]
- **Test Case Defect Density**, which describes the relation of tests that revealed defects with the total number of executed test cases [1]
- **Coverage Metrics**, which state the percentage of implemented code covered by test cases (see also Section 2.3.4) [1]

Note that all of the mentioned metrics are just examples out of the tremendous variety of available metrics. Hence, *code coverage metrics* in the context of dynamic analytical SQA, i.e., during testing activities, are of special interest for this thesis. Therefore, the following section will give more detailed fundamental knowledge in the field of *software testing* and corresponding metrics measured during this process.

2.3 Software Testing

The process of software testing includes all actions and activities conducted in order to execute a software system with the aim of comparing the observed behavior with its expected behavior [5]–[7]. More precisely, software testing means executing a program with the intention of finding errors. A *successful* software test is therefore a test that finds an error in the system [8], [14]. Tests that do not find any errors are considered as *unsuccessful* software tests [14]. Though this is the common definition of *successful* and *unsuccessful* tests in literature, those terms are usually used differently in practice. It emerged that tests that found errors are rather classified as *unsuccessful* tests and those that did not find any errors are referenced to as *successful* tests in practice [14]. For this reason the rest of this thesis will henceforth follow the latter, more practice-oriented understanding. In fact, an even more distinctive nomenclature will be used in order to have a clearer delimitation to the terms used in literature. Tests that did not detect errors will be referred to as *passed*, whereas tests that found errors will be stated as *failed* tests.

If tests do not find any errors, it is in general not implied that the system is error-free. For proving that a program does not have any errors or bugs, it would be necessary to test it under all possible system states and input variables, which is usually not possible in finite time (e.g., if the input variables are integers or strings of arbitrary length) [6], [8]. Furthermore, software tests are not intended to locate and resolve bugs. These activities are rather defined as debugging activities [6], [15]. In the upcoming section, a more specific definition of the terms *error*, *fault* and *failure* is given.

2.3.1 Errors, Faults and Failures

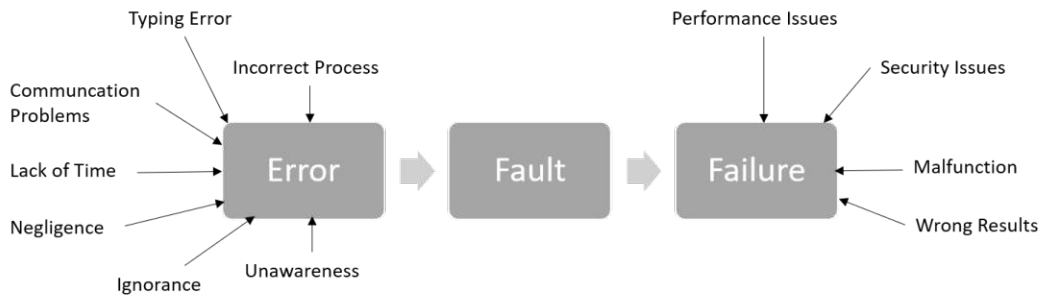


Figure 2.2: Errors, faults and failures [6]

An **error** is typically a human action that leads to incorrect outputs of a program, e.g., incorrect code due to a typing mistake. An error leads to a **fault**, which is an incorrect step, process or data definition in the program. A fault in a software system is often called a *bug*. A fault finally induces a **failure**, which is a deviation of the behavior of a software system from its expected behavior [6], [8], [15], [16]. Figure 2.2 shows the relation between those terms and states prominent examples for errors and failures.

More precisely speaking, software testing aims at finding errors in a software component or system in order to prevent faults and failures [6].

2.3.2 Test Levels

An important strategy for governing and maintaining software tests is dividing them into meaningful sets following to their concerns. The most common method is a breakdown of the tests into the following groups:

- **Component tests** are tests that try to find errors in separately testable components [8]. Those components are the smallest building blocks of the software that can reasonably be tested (e.g., classes, objects) [7]. Depending on the programming paradigm and language used (e.g., Java), component tests are often referred to as *unit tests* [6]. “Separately testable” means that the component under test must be isolated from other components in the software system, s.t. the behavior of the component can be tested completely without the influence of other components it depends on. This is achieved by establishing *mock objects* and *stubs*, which are placeholders for dependent components and simulate their behavior [6], [8]. Component tests are mostly written by the software developers themselves as they are developed at the same time as the components they should cover. Another very common strategy is writing the test code before the actual implementation of the component. The actual implementation of the component is driven by those tests

that actually assume and ensure an expected behavior. This strategy is known as *test-driven development* [7], [8].

- **Integration tests** are intended to test the interfaces between previously tested software components, the interactions between different system layers (e.g., operating system or file system) or cross-system interfaces [8]. The target is therefore to integrate components into larger subsystems and test the interaction with each other [6], [7]. Interfaces to foreign systems, the operating system or file system may need to be simulated, as they might not be available when the tests have to be executed. This is usually achieved similarly as in component testing by establishing mock objects [6].
- **System tests** focus on testing the specified behavior of the overall system with respect to customer requirements and expectations [6], [8]. Simulating other components is usually not necessary in this context, since all components and their interactions are available and tested through unit and integration tests. Hence, simulating interfaces to foreign systems (e.g., web services) could still be necessary [6]. The test environment should be set up and behave as close as possible to the production environment of the customer [8].
- **Acceptance tests** form the fourth level, are usually executed by customers or users of a system [8] and aim at verifying the implemented software system according to the requirements it must fulfill [6], [8]. In addition, acceptance tests may target non-functional requirements by e.g., executing performance or stress tests [5].
- **Regression tests** are a special form of software tests which get executed when extensions or changes are implemented in the system. The systems behavior may be changed through these actions and therefore it is necessary to re-run all or certain unit, integration and system tests [6], [8]. Regression tests are usually automated, as changes in the system (e.g., refactorings, code changes) are rather frequent. Automating those tests for the regular execution in the software life cycle saves time and is therefore economical [8].

2.3.3 Testing Strategies

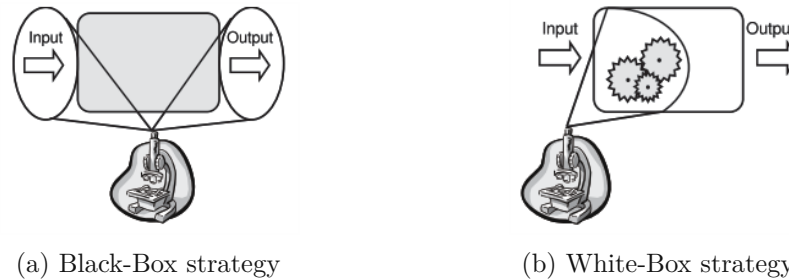


Figure 2.3: Comparison of black-box and white-box testing strategies [6]

Testing strategies are distinguished between **white-box** and **black-box strategies** in literature [5], [8], [15], [17]. Figure 2.3 shows a schematic comparison of those two concepts. White-box testing considers the inner structure of the concrete implementation which is the strategic basis for test case selection and implementation [8], [17]. The detailed knowledge of the implemented code and structure is therefore necessary and the target is to cover code sequences considering different coverage levels (see also Section 2.3.4). White-box testing therefore also offers the capability of locating errors in the source code as the concrete statements, conditions and paths in the program are tested [6]. Black-box tests, on the other hand, do not consider the inner structure of a program and are usually derived from a system's or program's specifications or requirements [15], [17]. Test objects are therefore tested independently from their implementation and get called with defined input parameters. The outputs are then compared with the expected output of the test object [6].

Due to the high relevance for this thesis, the following Chapter 2.3.4 will especially describe *white-box* testing methods, which are also referred to as *structural testing methods* [17].

2.3.4 Structural Testing Methods

The main target of structural testing methods is identifying paths in the program's control flow graph and establishing tests that cover them [1], [8]. A possible control flow graph for a given Java method is depicted in Figure 2.4.

The term **coverage** is defined as the percentage of source code that has been reached by tests in relation to the code that is available [1]. Structural testing distinguishes between 5 levels of coverage, that are connected to the control flow graph of a program:

- **Statement Coverage** or C_0 -Coverage means that all nodes in the control flow graph are covered at least once, which implies that every statement of the program is executed at least once. This level is considered as the weakest level because important branches or paths in the control flow graph may be missed. [1], [8], [15].

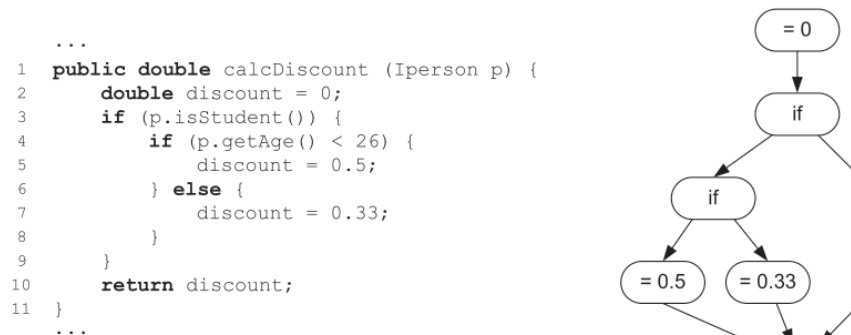


Figure 2.4: Control flow graph example [6]

- **Branch Coverage** or C_1 -Coverage aims at covering each branch and therefore all possible ways in the control flow graph. This means in particular, that all conditions (e.g., if-conditions) in the program flow must be evaluated at least once both to *true* and *false*. Furthermore, all case- and switch-statements need to be tested in all variations, as well as exception handling blocks. Note that if a condition is composed of atomic conditions and logical operators (e.g., the condition $x == 4 \parallel y == 5$), only the composed condition must be considered and therefore evaluate to *true* and *false* [6], [15].
- **Simple Condition Coverage** or C_2 -Coverage enhances C_1 -Coverage to the extent that also the atomic conditions of each branch need to be evaluated at least once to *true* and *false*. This is a much more detailed coverage than the latter, but considers specifically the atomic operations without any respect to the overall condition for a branch. This could lead to the effect that not all branches get covered. Furthermore, C_2 -Coverage leads to *error masking* in certain cases, e.g., if conditions composed with *OR*- or *AND*-operators are considered. *OR*-conditions evaluate to *true* if one side evaluates to *true*. If the other side is e.g., a method call which leads to an error, this would not be recognized by the test case [6], [8], [15].
- **Branch Condition Combination Coverage** or C_3 -Coverage tries to evaluate all combinations of the atomic conditions and the composed conditions. This results in a very high testing effort, as n conditions result in 2^n test cases. A weakened variation of this level is *Modified Branch Condition Testing*, which only considers combinations of atomic conditions that result in a different result for the overall conditions [6], [8], [15].
- **Path Coverage** or C_4 -Coverage aims at covering all possible paths in the control graph. A path is a unique sequence of conditions and statements. In addition to other coverage levels, this leads to the ability to also consider loops in the program.

Hence, this results in a rapid growth of the possible paths. Loops without a fixed iteration boundary may even result in infinitely many execution paths. Therefore, the practical value of C_4 -Coverage is per se very low, but may also bring a surplus if its usage is well considered (e.g., if the program can be feasibly tested in this way) [1], [6], [8], [15].

2.4 Refactoring

A refactoring is characterized as a change made to the internal structure of a software with the intention of making it easier to understand and cheaper to modify without changing its external and observable behavior [18], [19]. The process therefore involves activities like the removal of code duplication, the simplification of complex logic and the clarification of unclear code [20]. The following abstract example will give a more precise definition.

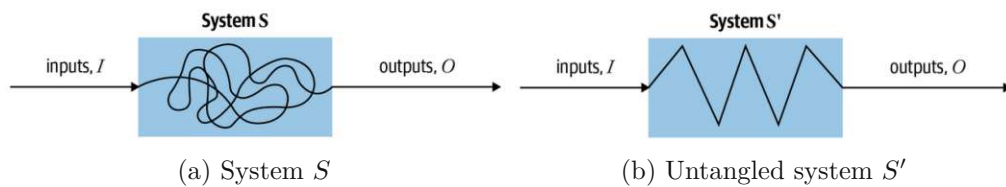


Figure 2.5: Refactoring: Abstract Example [19]

Consider a software system S , which inner structure is heavily tangled up and has defined inputs I and outputs O [19] (Figure 2.5a). Refactoring activities would now aim at transferring the system S to a successor S' , which inner structure is untangled, but hence delivers the same outputs O for given inputs I and therefore has the same expected behavior from a consumer's view [19] (Figure 2.5b).

In order to ensure that the system behaves the same after refactoring activities have been conducted (i.e., that the outputs O stay the same for given inputs I), it is an essential precondition to have solid and automated tests [18], [20]. The latter will furthermore lead to more courage to refactor and more willing to try experimental designs as they establish confidence that the refactored system still behaves equally to the initial version [20].

In the following sections, the benefits and problems of refactoring activities are further enlightened.

2.4.1 Benefits of Refactoring

In the first place, regular refactorings support in maintaining, preserving and improving a programs design, as well as keeping the code understandable [18]–[20]. As software evolves and code is added or changed, the code will lose its structure, which makes it

harder to understand, extend and maintain. In more detail, continuous refactorings especially provide i.a. the following benefits.

Refactoring typically includes **eliminating duplicated code**, which is very important for future modifications. The more code there is, the harder it is to understand and modify. Furthermore, eliminating the duplicates ensures that the code says everything once and only once, which is an inevitable factor of a good design [18].

Apart from the fact that good design leads to better maintainability of code, refactoring activities support the process of **design preserving feature and code enhancements**. In detail, refactoring in this scope includes the analysis and execution of changes in an existing software that enable better accommodation of new features and code instead of implementing an enhancement without regard to how well it fits with an existing design [20].

Moreover, a well-maintained codebase can significantly decrease the effort and time needed for **identifying bugs** [18], [19]. On the one hand, breaking up complex statements into smaller pieces and extracting logic into new functions can both imply a better understanding of what the code is doing and isolate potential bugs. On the other hand, regular refactorings during the development process can make it easier to spot and avoid bugs as early as possible [19].

Summarized, refactoring and its positive impacts as described in this section lead to **higher productivity** in the software development process [18], [19]. Though refactoring activities are additional efforts taken, a good design is essential for rapid software development. Without a good design, it may be the case that developers can progress quickly in the beginning. Hence, they will spend more time later on during the previously described activities if the design is bad. In addition, refactoring stops the design of a system from decaying and can even improve it [18].

2.4.2 Problems of Refactoring

First, it needs to be repeated that in the context of a refactoring it has to be ensured, that the behavior of the system remains identical. The confidence that this is the case can be increased by establishing a suite of tests with sufficient coverage before a refactoring is conducted. Hence, even with thoroughly implemented test cases, there is still the chance that there are certain **edge cases not covered by the tests** [19]. In worst cases, refactorings may also unintentionally **reveal dormant bugs** in the system [18].

Moreover, there are also refactoring activities that come along with **changes concerning external interfaces** (e.g., renaming a method). If developers have access to all spots where the interface is used, then this is not a problem in the first place. Nevertheless, there is a problem if the interface is used by consumers and systems that are out of scope of the system to refactor. Such refactoring steps are therefore more complicated and need special attendance in order to not impair the observable behavior of the system (e.g., by keeping the old and the new interface until consumers reacted to the change) [18].



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

State of the Art

The following sections include the main outcomes of the conducted research phase of this thesis. In Section 3.1, current research concerning information needs in software engineering, as well as scientific approaches for fulfilling them are summarized. Furthermore, this section reviews scientific work in the area of the assessment of test suite quality and effectiveness. Section 3.2 presents features in currently available code coverage and unit testing tools that support stakeholders with fulfilling the found information needs by transferring the stated scientific approaches into tools that are used in practice. Finally, Section 3.3 consolidates the findings.

3.1 Related Work

Researchers have been conducting numerous surveys in order to identify prominent information needs of different stakeholders in software projects and to make statements about the extent to which they are fulfilled. Section 3.1.1 summarizes diverse surveys and studies that had a significant influence on this thesis and lead to the initial ideas and considerations. Section 3.1.2 reviews scientific approaches for fulfilling the found information needs in Section 3.1.1. Moreover, Section 3.1.3 summarizes the work of Bowes *et al.*, who did important research on inevitable testing principles and how they can be quantified in order to assess a test suite's quality and effectiveness.

3.1.1 Information Needs in Software Engineering

Begel and Zimmermann came up with a ranked list of 145 questions that software engineers want to have answered by data scientists in software projects [22]. The following two questions, which were ranked beneath the top 20, indicate that there is a high interest concerning the impact and risks of code changes, as well as tool support in this area:

- “*What is the impact of a code change or requirements change to the project and tests?*”
- “*What tools can help us measure and estimate the risk associated with code changes?*”

Another study conducted by Kim *et al.*, which specifically dealt with analyzing refactoring challenges and benefits, affirms the latter perception [23]. In particular, the researchers asked their participants to state what the challenges associated with a refactoring are in their perspective, whereby 28% of the participating developers mentioned inherent challenges such as the difficulty of ensuring program correctness after a refactoring. Furthermore, 29% of the participants also mentioned that there is a lack of tool support for refactoring change integration and code review tools targeting refactoring edits. As a conclusion, the researchers proposed further research in this direction, as there is the need for various types of refactoring support that goes beyond the given automated refactoring mechanisms within an Integrated Development Environment (IDE) [23].

Tao *et al.* furthermore investigated on the practices used in order to determine a change’s risk and which information needs developers have during those tasks. In the context of this thesis, the following three questions that aroused during the conducted survey are of interest [24]:

- “*Which test cases should be run to verify this change?*”
- “*Is any additional test case needed to cover this change?*”
- “*Which part of the change may cause the test case(s) to fail?*”

In addition, the researchers concluded that two approaches are typically used to check whether a change breaks the applications functionality or not, namely *unit and regression testing* and *manual code reviews and inspections* [24]. The first one is considered as highly time consuming and furthermore dependent on the adequacy and quality of the established test suite. The second one requires tremendous manual effort (e.g., for checking all dependencies of changed parts) and good support of the compiler, debugger and static analysis tools. Most remarkably, one of the survey’s participants explicitly stated the following [24]:

“None of those (practices) are really very satisfying, though, as my confidence level in the change is not as high as I would like.”

The participants therefore called for explicit features that detect code portions impacted by a change and the affected test cases.

Hence, the surveys found during the research phase of this thesis not only covered information needs in the field of code changes and refactorings. The researches conducted by Begel and Zimmermann, Fritz and Murphy and Bowes *et al.* furthermore revealed

interesting questions that deal with information needed for understanding the correlation between source code and tests, as well as for evaluating and assessing the quality of test suites [21], [22], [25]:

- “*How do test cases relate to packages/classes?*”
- “*How should we handle test redundancy and/or duplicate tests?*”
- “*How good are our tests?*”

Concerning the latter question, Bowes *et al.* furthermore presented important testing principles and guidelines, that will also be reflected later in Section 3.1.3.

Moreover, it needs to be held that various surveys also revealed that (apart from code- and test-specific information) co-worker awareness and process-oriented information are very prominent areas in which it seems to be hard to gain desired information reasonably. Specifically, the examined surveys stated recurring questions as follows [25]–[27]:

- “*What have my co-workers been doing?*”, “*Who is working on what?*”, “*Who is working on the same file as I am?*”
- “*How have resources I depend on changed?*”, “*What is the recent activity on a plan item?*”, “*Which features and functions have been changing?*”
- “*How is the team organized?*”, “*Which conversations in work items have I been mentioned?*”, “*Whom to talk to regarding a particular project or file?*”

3.1.2 Fulfilling Information Needs

Researchers have also been trying to fulfill information needs stated in the latter section by establishing various concepts and tools. During the research phase of this thesis, especially contributions in the areas of *co-worker awareness*, *code change understanding* and *test redundancy detection* have been found. The following sections consolidate those approaches.

Co-worker Awareness

When considering information needs in the context of *co-worker awareness*, there has been remarkable work both in research and prototype development in order to relieve the high communication effort in this area.

For example, Fritz and Murphy introduced an *information fragment model* [25] that supports composition and presentation of information from various source areas in order to answer those types of questions. Furthermore, Sharma and Kaulgud presented the Project Insights and Visualization Toolkit (PIVoT) [28], which classifies in-process data

coming from different tools, overlays relationships between the data elements and provides a rich set of analysis which can provide composite metrics and insights in the dimensions of the process and activities, development quality and team dynamics. Other remarkable examples for tools that focus especially on building co-worker awareness during software development processes are FASTDash [29] and Palantír [30].

Code Changes

Apart from research in supporting developers in the area of *co-worker awareness*, there are also attempts to satisfy information needs in the context of *code changes*. In fact, there are certain researchers that investigate especially on the concepts of *change impact analysis* and *change-driven testing*.

Change impact analysis tries to estimate what the affected artifacts of a software will be if a software change is made [31]. It consists of a collection of techniques for determining the effects of source code modifications, which allows programmers to experiment with different edits and find the code fragments and tests that they affect [32].

Change-driven testing is related to the latter and leverages the concepts of *test-impact analysis* and *test-gap analysis* [33]. The first is intended to automatically find the relevant tests for any given code change, as well as sorting them in a way that increases the possibility of finding mistakes introduced early on. The second one should identify test gaps, i.e., code changes that lack tests.

The time and effort needed for augmenting test suites may therefore be remarkably reduced as the information can be used to determine which test cases should be added in order to cover a respective change. Furthermore, the time needed for running regression tests can be shortened as it can be determined which test cases are definitely not affected by the change [32]. Researchers therefore proposed and evaluated various tools that make use of those concepts and try to assist developers and testers in gaining higher productivity by relieving regression testing and test suite augmentation effort.

For example, Amann and Jürgens illustrated the use of *change-driven testing* by augmenting the Continuous Integration (CI) pipeline for a specific software product with additional profilers and test-gap tree maps [33]. Furthermore, Wloka *et al.* presented the change-aware unit testing tool *JUnitMX*, which is an extension to the JUnit Eclipse plugin and leverages change impact analysis to guide developers in writing more effective unit tests. Ren *et al.* moreover proposed and evaluated *Chianti*, which is a remarkable change-impact analysis tool for Java implemented in the context of the Eclipse environment [32], [35].

Test Redundancy Discovery and Test Suite Reduction

Moreover, there exist works that address test redundancy measurement and discovery in particular. Test redundancy may be either of syntactic or semantic nature. Syntactic test redundancy is similar to normal code duplication and may be discovered through

simple static code analysis tools and eliminated by refactoring methods (e.g., method extraction, etc.) [36]. Semantically redundant tests are characterized as such when they do not improve the fault detection capability of the test suite [36], [37]. The latter are harder to detect, which is why developers and testers need special assistance in order to discover them.

Researchers have therefore been working extensively on test redundancy discovery and especially on test suite reduction. Most approaches in this research field build upon the idea of finding a subset of test cases (i.e., a representative test set) that still fulfills all test requirements of the original test suite. This problem is in theory classified as *finding the minimum cardinality hitting set* [38], [39] which is an NP-complete problem [40]. The research works encountered therefore propose different variations of heuristics and coverage criterions in order to find an optimal minimum test set. The most notable approaches are those of Harrold *et al.* [38], Rothermel *et al.* [39], Wong *et al.* [41], Jones and Harrold [42] and Offutt *et al.* [43].

Hence, it has been shown that approaches of this type generally reduce the fault detection effectiveness [37]. Fraser and Wotawa have therefore proposed a different approach by using model-checker techniques and Kripke structures, which does not have a negative influence on the fault detection ability of a test suite [44]. Moreover, Koochakzadeh *et al.* also stated that redundancy detection could be improved by applying more coverage criteria and by using more precise coverage tools [37].

In addition, Koochakzadeh *et al.* have found an interesting feature for identifying test redundancies in the coverage tool *CodeCover*, which will be presented later in Section 3.2.

3.1.3 Assessing Test Suite Quality and Effectiveness

Testing by itself is a very important activity in ensuring software quality as software tests form safety nets when modifying productions code [21], [36]. However, their quality is usually taken for granted and overlooked. It is definitely possible to fulfil certain criteria (e.g., a certain level of code coverage) and still not having high quality tests that are able to verify and/or break the system in a sufficient and effective way [21].

Bowes *et al.* did important research on this topic that aimed at identifying important testing principles and discussing how they can be quantified for assessing the goodness of tests. They conducted workshops with industry partners, where they discussed and brainstormed ways to address and evaluate the quality of tests. Afterwards, they merged the outcome with their experience and existing literature and came up with a final list of 15 best practices for unit test design [21]. The following listing summarizes selected practices with a high correlation to one of the key ideas of this thesis, namely the question on how further processing of coverage metrics and coverage density can help assessing the code coverage and coverage distribution of a project (i.e., a test suites quality). Note that the principles are also related to common test smells, e.g., as analyzed by Deursen *et al.* in their research work on this topic [36].

Testing behavior (not implementation)

Bowes *et al.* proposed that there should be more focus on testing the expected behavior of a program rather than its implementation. Testing implementation details makes tests more dependent on a particular implementation of specifications, which implies a higher effort for updating a test implementation on a new program implementation [21]. On the other hand, keeping focus on the expected behavior and utilizing public interfaces of the System under Test (SUT) would avoid such problems. The researchers also stated that it is relatively easy to reach high coverage without testing expected behavior. Therefore, the interpretation of coverage metrics as a sign of test quality should be avoided, as there is research questioning the relation with test effectiveness [45]–[47]. Nevertheless, high coverage yet will be achieved with a focus on testing behavior [21].

Single Responsibility

One test should have a single reason to fail, which implies that it should always be possible to locate the root cause of a test fail [21]. Bowes *et al.* defined this principle in the context of the number of assertions per test and propose to strictly enforce verification of one condition per test. In a broader sense, this principle also refers to the fact, that e.g., one unit test case should not test more than one unit at the same time, which gets also emphasized by their proposal of using the count of unique method calls for the assessment [21]. If there are calls to multiple methods in the class under test, this might indicate that the test is testing more than it should in a single case.

Importance of Maintainability of Test Code

As production code evolves, it is inevitable that test code will evolve too [21]. Maintainability of test code should therefore always be a key goal and especially duplicated and redundant test cases should be avoided at all costs.

Tests should not dictate the Code

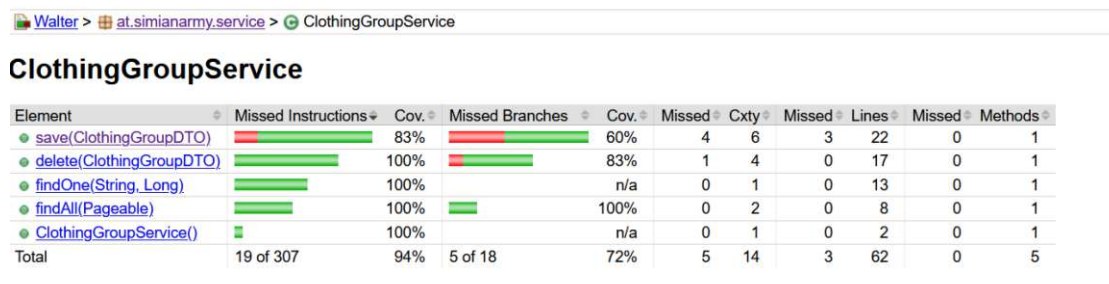
This principle suggests that there should not be any modifications in production code for the purpose of testing [21]. Examples for such modifications are to update access modifiers of production code methods in order to reach them from the test code (e.g., from private to public) or to write methods that only get accessed by test code to get internal states of the class under test.

3.2 State of the Art Tools

In this section, currently available tools for measuring and visualizing code coverage that are in touch with the related work of this thesis are presented. Note that the focus is especially on the question whether and how they fulfill the revealed information needs summarized in Section 3.1.1 and support developers for reaching compliance of testing suites with the testing principles presented in Section 3.1.3. Note that only code coverage tools for software implemented in Java and tested with the JUnit framework have been reviewed as its environment offers a comprehensive set of testing tools. Moreover, testing tools for other popular programming languages do not reinvent the wheel and mostly offer similar features to those in common tools for Java.

3.2.1 JaCoCo

*JaCoCo*¹ is a free code coverage tool for Java, which generates both HTML and XML reports. The latter can be easily integrated and reused in other tools, such as build servers (e.g., Jenkins). Furthermore, many common IDEs such as Eclipse and IntelliJ provide integration of the reports in their platforms. The generated reports consist of visualizations on different abstraction layers. On the one hand, coverage information is presented per package and class in tabular form (Figure 3.1). On the other hand, the information can be viewed on the level of LOC for each class (Figure 3.2).



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
save(ClothingGroupDTO)		83%		60%	4	6	3	22	0	1
delete(ClothingGroupDTO)		100%		83%	1	4	0	17	0	1
findOne(String, Long)		100%		n/a	0	1	0	13	0	1
findAll(Pageable)		100%		100%	0	2	0	8	0	1
ClothingGroupService()		100%		n/a	0	1	0	2	0	1
Total	19 of 307	94%	5 of 18	72%	5	14	3	62	0	5

Figure 3.1: *JaCoCo* HTML Report: Coverage Information

As depicted in Figure 3.1, the tool lists and presents the reached coverage on different levels, such as instruction, branch, line and method coverage. Note that in *JaCoCo*, *instruction coverage* refers to Java byte code instructions and provides information about the amount of code that has been executed or missed. *Line coverage* refers to implemented LOCs, which may compile to multiple byte code instructions [53].

```

48.     public ClothingGroupDTO save(ClothingGroupDTO clothingGroupDTO) {
49.         log.debug("Request to save ClothingGroup : {}", clothingGroupDTO);
50.         ClothingGroupDTO result = null;
51.         ClothingGroupDTO db = findOne(clothingGroupDTO.getCsize(), clothing
52.
53.         if (clothingGroupDTO.getQuantity() == 0) {
54.             clothingGroupDTO.setQuantity(0);
55.         }
56.         if (clothingGroupDTO.getQuantity() < db.getQuantity()) {
57.             clothingGroupDTO.setQuantity(db.getQuantity() - clothingGroupDT
58.             result = delete(clothingGroupDTO);
59.
60.         } else if (clothingGroupDTO.getQuantity() > db.getQuantity()) {
61.             clothingGroupDTO.setQuantity(clothingGroupDTO.getQuantity() - d
62.             if (clothingGroupDTO.getQuantity() != null) {
63.                 Integer i = clothingGroupDTO.getQuantity();
64.
65.             ClothingType clothingType = clothingTypeRepository.findById
66.
67.             while (i != 0) {
68.                 Clothing toSave = new Clothing();
69.                 toSave.setSize(clothingGroupDTO.getCsize());

```

Figure 3.2: *JaCoCo* HTML Report: Class View

¹<https://www.jacoco.org/>

3. STATE OF THE ART

Figure 3.2 shows a method in *JaCoCo*'s class view, which is accessible from the coverage information list (i.e., by clicking on the respective element in the first column). In this visualization, code coverage information is highlighted using two different methods. On the one hand, the line colors state whether the respective line is covered (green) or uncovered (red). A yellow line highlighting indicates that the line has been covered, but not all byte code instructions of that line have been hit. On the other hand, branch coverage is indicated by using diamonds with different colors, whereby a green diamond indicates that all branches have been covered. In analogy, yellow and red diamonds indicate partial or no branch coverage.

3.2.2 Cobertura

*Cobertura*² is a similar tool to *JaCoCo*, but hence outdated and no longer actively maintained [54]. Furthermore, it does not support current Java versions. It visualizes coverage information very similar to *JaCoCo* in separate tabular listings for classes and packages (Figure 3.3), as well as class views.

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	77	60%	41%	1,26
at.simanmv.ace.logging.annotation	1	N/A	N/A	0
at.simanmv.config	21	62%	35%	1,659
at.simanmv.config.filestore	1	73%	73%	6
at.simanmv.domain.util	3	42%	N/A	1,316
at.simanmv.export.api.util	1	N/A	N/A	1,091
at.simanmv.filestore.service	1	40%	14%	1,571
at.simanmv.imports.accounting.csv	2	100%	N/A	1
at.simanmv.imports.data	1	0%	N/A	4
at.simanmv.multitenancy	1	74%	80%	2
at.simanmv.notification	1	N/A	N/A	1
at.simanmv.results	1	18%	N/A	2,5
at.simanmv.repository	18	N/A	N/A	1
at.simanmv.scheduler	6	60%	N/A	1
at.simanmv.scheduler.task	9	76%	35%	2,111
at.simanmv.security	3	50%	50%	1,5
at.simanmv.service	2	33%	N/A	0
at.simanmv.service.util	3	24%	0%	1,714
at.simanmv.web.controller	1	0%	0%	2,323
at.simanmv.web.view.errors	1	0%	N/A	1

Report generated by Cobertura 2.1.1.1 on 03.01.22 17:49.

Figure 3.3: *Cobertura* Coverage Report

Though this tool is outdated and cannot be used with newer Java versions, it provides one important extension that its competitor *JaCoCo* does not offer. In fact, while *JaCoCo* only states whether a line is (partially) covered by unit tests or not, *Cobertura* also visualizes how often a LOC is covered in its class view (Figure 3.4).

```

48 public List<String> getLocationsToInit() {
49 2 List<String> result = new ArrayList<>();
50
51 2 Set<String> tenants = this.tenantConfigurationJsonRepository.getAllTenantIds();
52
53 2 if (!this.walterProperties.getMultitenancy().isEnabled()) {
54
55 2 result.add(this.fileStoreProperties.getTemplatesLocation());
56 2 result.add(this.fileStoreProperties.getDocumentsLocation());
57 2 result.add(this.fileStoreProperties.getReceiptsLocation());
58
59 } else {
60 0 for (String tenant : tenants) {
61 0 result.add(tenant);
62 0 result.add(tenant + "/" + this.fileStoreProperties.getReceiptsLocation());

```

Figure 3.4: *Cobertura* Coverage Report: Class View

²<https://cobertura.github.io/cobertura/>

3.2.3 CodeCover

*CodeCover*³ is also very similar to *JaCoCo* and *Cobertura* in its basic functionality. Hence, as also stated by Koochakzadeh *et al.*, it offers a very interesting additional feature for inspecting tests concerning possible test case duplication [37]. For this, the tool provides a *correlation view* as depicted in Figure 3.5.

The visualization is separated into a tree view on the left side and a test case matrix on the right side. The first displays similar test cases for each test case in a hierarchy, whereby “similar” means that those tests cover the same parts of the code that the initial test case covers [55]. The matrix provides information about the extent to which the test cases resemble each other pairwise. This is emphasized by colors for each pair, whereby the colors state resemblance categories (e.g., red = 0%-33% resemblance, blue = 33% - 66% resemblance, etc.).

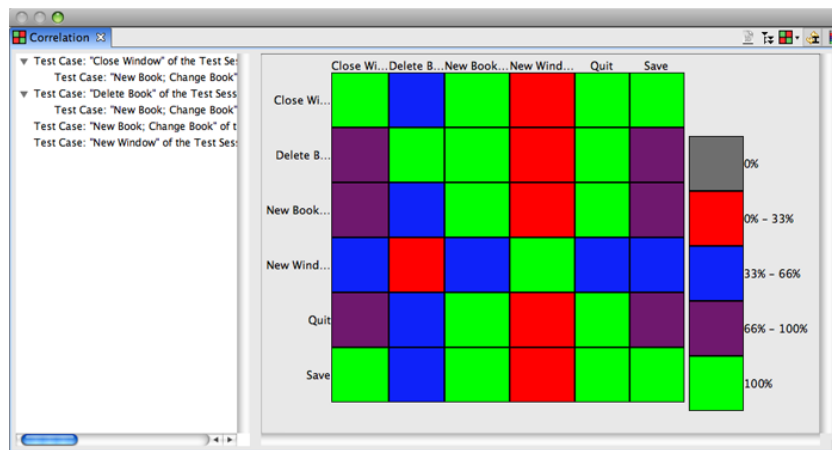
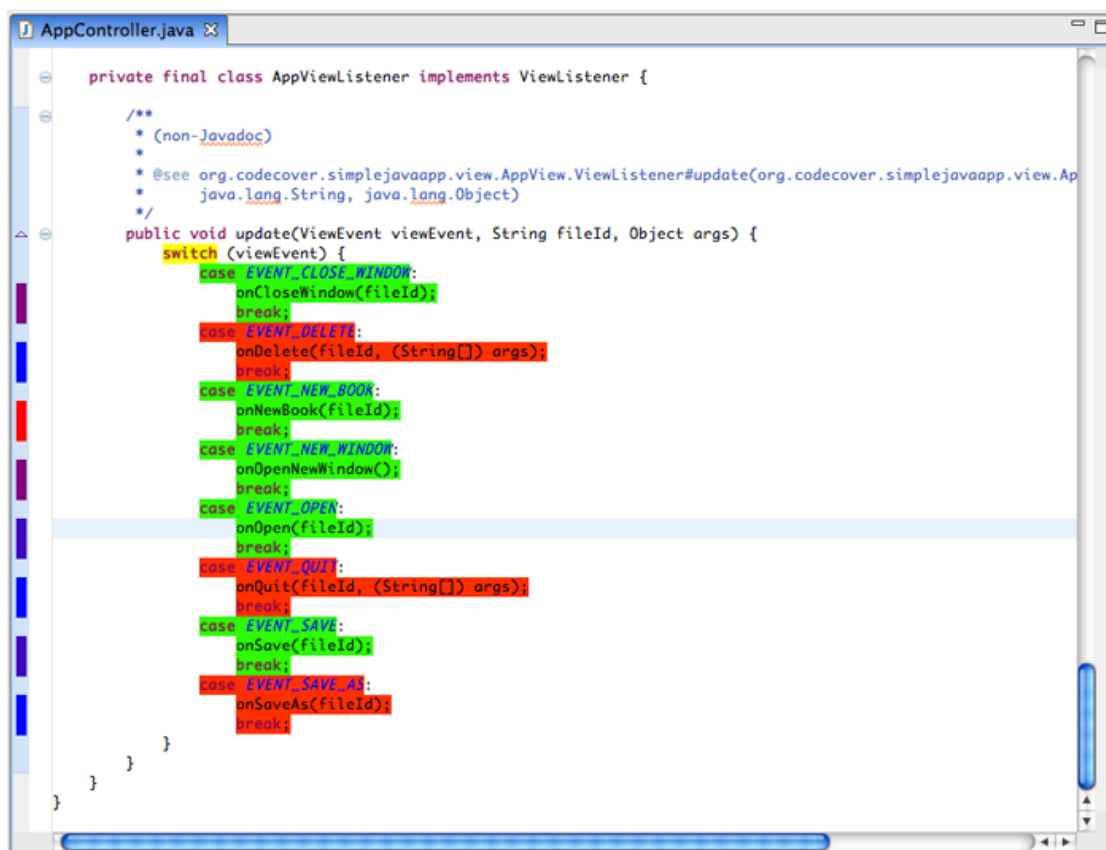


Figure 3.5: Correlation View in *CodeCover* [55]

Moreover, the tool provides source code highlighting based on the coverage frequency measured (Figure 3.6). On the one hand, the degree to which a line is covered is visualized in a similar way to other tools (i.e., green = covered, red = uncovered). On the other hand, the *hot path* of the program is depicted on the left side of the editor with small colored rectangles, whereby the more often a statement was executed, the redder the rectangle becomes [55]. In addition, there is the possibility to show test cases that cover a certain part of the code.

Though those features provide real added value to classical coverage analysis, the downside is that *CodeCover* has not been updated since 2011 and does not seem to be maintained anymore. Furthermore, it is rather usable as a command line tool because IDE integration is only given for Eclipse. Hence, due to the lack of maintenance, it seems that the integration does not work with current versions of Eclipse as encountered while evaluating the tool for this thesis.

³<http://codecover.org/>

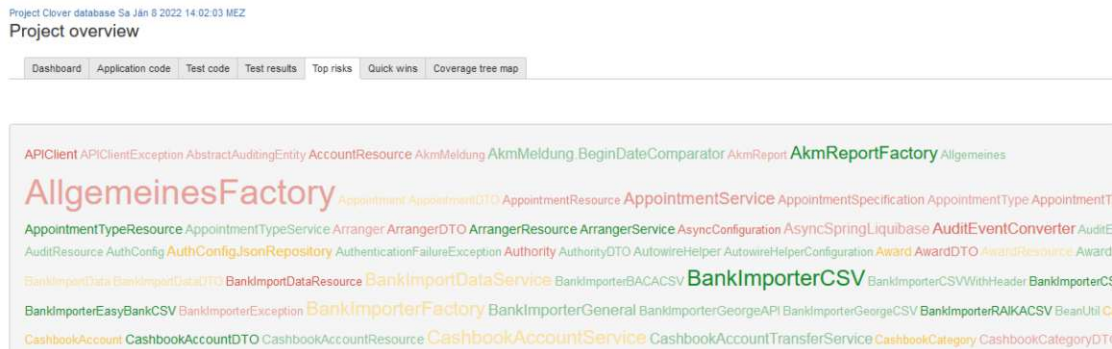
Figure 3.6: Code Highlighting and Hot Path in *CodeCover* [55]

3.2.4 OpenClover

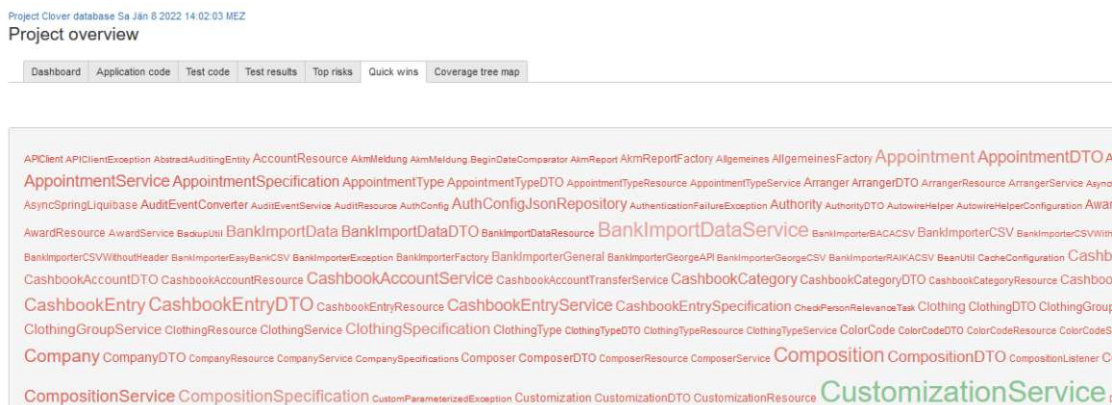
*OpenClover*⁴ is another open-source tool, which was created as a fork from Atlassian's formerly commercial tool *Clover*. Apart from similar basic features that also *JaCoCo* and other tools provide (i.e., visualization of code coverage metrics in tabular listings and class views), its main strength lies in combining static code metrics, complexity metrics and code coverage metrics in order to provide high sophisticated dashboards for assessing the quality of the test suite.

One of those dashboards is the overview of the project's top risks (Figure 3.7). The visualization is a tag cloud, whereby the tags state classes within the project. The cloud highlights those, that are most complex but least covered by the implemented test cases. The font size represents the average method complexity metric and the color states the total coverage (whereby red means 0% and green 100% coverage). The classes that represent the greatest risks for the project in terms of complexity and their respective coverage are therefore those that have the biggest font sizes and are colored in red.

⁴<https://openclover.org/>

Figure 3.7: *OpenClover* Top Risks

Moreover, the tool provides a tag cloud with different semantics in order to identify quick wins for the projects code coverage (Figure 3.8). The font size represents the total number of elements the respective class has (number of statements + number of branches + number of methods), while the color represents the number of tested elements - again with color gradations from red (low coverage) to green (high coverage). The greatest increase in code coverage may therefore be achieved by covering the largest and reddest classes first, as they contain the highest number of untested elements.

Figure 3.8: *OpenClover* Quick Wins

Another visualization that stands out is the provided coverage tree map, which enables simultaneous comparison of classes and packages by their complexity and reached code coverage (Figure 3.9). The clusters state packages, whereby the sections within those represent their classes. The size of the sections states the measured complexity of the respective class. Similar to the latter visualizations, the colors indicate the reached code coverage.

3. STATE OF THE ART

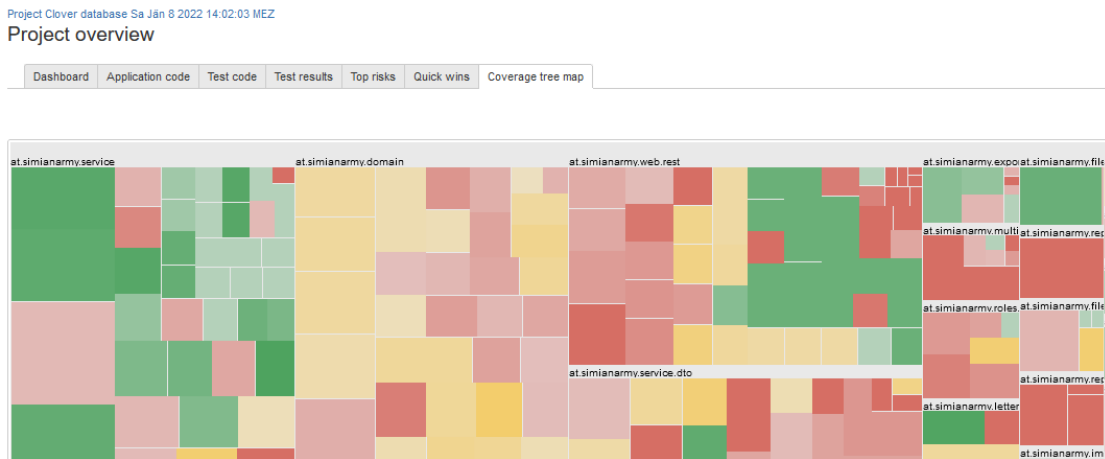


Figure 3.9: *OpenClover* Coverage Tree Map

In addition and similar to other tools described earlier, the code coverage can be analyzed in a class view, where the LOCs are highlighted in green/red for stating that the respective line is covered/uncovered. The counter next to the line numbers indicates how often the line is covered by tests. By clicking on the line counter, it is also possible to view which tests cover the respective line.



Figure 3.10: *OpenClover* Class View

It also needs to be held that clover offers high sophisticated and highly reusable XML reports which contain the recorded coverage data for a project in machine readable format. Furthermore, a database is generated during code instrumentation and test execution, which contains structured information on which tests cover which LOC. The latter can specifically be read and reused for further processing with the aid of *OpenClover's* Java library⁵, which provides functionality for accessing and reading the instrumentation database.

3.2.5 Parasoft Jtest

The tool *Parasoft Jtest*⁶ is a commercial tool that stood out in particular during the research phase of this thesis due to its specialized features for change-driven testing and test implementation guidance. In particular, the tool offers the following functionality:

- Static code analysis, e.g., pattern-based analysis, code metrics (i.e., code complexity analysis) and code duplication analysis, as well as analysis of compliance with security standards
- Artificial Intelligence (AI) assisted features such as
 - Automatic unit test creation
 - Test suite maintenance recommendations
 - Automatic identification of code gaps (untested code) and suggestions on how to cover them
- Code coverage information collection during unit testing and runtime of real applications
- Code coverage reporting, which also provides traceability between unit tests and source code
- Test impact analysis, i.e., identifying and executing tests that need to be run in order to validate code changes
- Association of tests with requirements (by relating them with data gathered from external systems, e.g., Jira) for verifying which capabilities of the software have been tested and understanding the impact of test failures across requirements and user stories
- Integration in various other tools, such as build tools, IDEs and CI servers

⁵<https://github.com/openclover/clover/tree/master/clover-core>

⁶<https://www.parasoft.com/products/parasoft-jtest/>

Note that the provided features and functionalities listed above were not examined during the research phase. The reason for that was, that a free trial of the software cannot be obtained via Parasofts homepage and it would have been necessary to contact the distributor in order to get a trial license. Therefore, information in this section is based solely on the service description and fact sheets offered on Parasofts homepage [56], [57].

3.3 Summary

As described in Section 3.1.1, there are various information needs that have been revealed through diverse studies and investigations. Summarized, it emerged that participants need support for fulfilling those especially during the following activities (note that henceforth the identified information needs will be referenced to as *IN1* - *IN7*):

- *Information Need 1 (IN1)*: Gaining a general understanding on the impacts of code changes
- *Information Need 2 (IN2)*: Identifying tests that need to be run to verify a code change
- *Information Need 3 (IN3)*: Deciding whether there are additional test cases needed in order to cover a code change
- *Information Need 4 (IN4)*: Assessing the risk of code changes
- *Information Need 5 (IN5)*: Analyzing how tests and source code are related
- *Information Need 6 (IN6)*: Assessing the quality of a test suite
- *Information Need 7 (IN7)*: Identifying test redundancies

The revealed information needs in the area of co-worker awareness were not pursued any further, as the context and idea of this thesis is clearly of technical nature and deals with implementation and testing activities rather than software development processes and interpersonal communication.

3.3.1 Reflection of the Findings

This section will now once more reflect at a glance how scientific approaches and currently available tools fulfill those information needs and furthermore state where support is clearly missing.

Impact of Code Changes (*IN1-3*)

The conducted research showed that there already has been remarkable scientific work in this area by investigating on approaches like *change impact analysis* and *change-driven testing*, that, on the one hand, assist in gaining more understanding on the impacts of code changes (*IN1*) with reference to which other code fragments and especially tests are affected by a code change (*IN2*) and, on the other hand, in deciding if there are additional test cases needed (*IN3*). Furthermore, the concepts and ideas have been transferred into features offered by tools implemented for scientific (e.g., *JUnitMX* and *Chianti*) and practice-oriented (industrial) intentions (e.g., the commercial testing tool *Jtest*).

Assessment of the Risk of Code Changes (*IN4*)

The found concepts and features in the context of *change impact analysis* and *change-driven testing* do not state a valuable support for assessing the risk of a code change. The approaches and tools rely on the fact that the change to be analyzed has already been made and investigations are conducted after this activity. Hence, assessing the risk of a change after it has been made is especially not reasonable for refactorings (which is a special form of a code change) as they usually need a prior estimation on whether they can be done safely or further test cases need to be added (see also Section 2.4).

Understanding the Relation between Code and Tests (*IN5*)

Some of the evaluated code coverage tools offer fundamental support in this area by showing how often a line is covered and which tests cover them (e.g., *Cobertura*, *CodeCover* and *OpenClover*). Hence, this information is given only on the lowest level (i.e., on the level of LOC in class views), has rather low importance in contrast to other features offered and is not reused in further computations and visualizations on higher abstraction levels. Nevertheless, reusing this information in other scopes could further support in understanding how the test cases spread across classes or packages.

Assessing the Quality of a Test Suite (*IN6*)

Providing support concerning the decision where code coverage should be increased as provided by the high sophisticated tag cloud visualizations in *OpenClover* is only one side of the coin concerning the assessment of a test suites quality. In relation to the testing principles depicted in 3.1.3, it is furthermore inevitable to clearly understand how tests and code relate in order to gain consistency with those principles. The evaluated code coverage tools do not attribute enough importance to this, though it would be desirable that such tools put more emphasis on the semantic relationships between test cases and covered code.

Identifying Test Redundancies (*IN7*)

As described in Section 3.1.2, there has been extensive research effort on the topics *test suite reduction* and *identifying test redundancies*. All of the encountered scientific approaches deal with those problems on a theoretical level by trying to reduce test suites and find redundancies with test suite minimization algorithms and model-checker techniques. On the one hand, it is proven that especially methods of the first type generally reduce the fault detection effectiveness. On the other hand, test redundancy analysis does not seem to be a key aspect of currently available code coverage tools. The only tool that provides visualizations in this area, but hence is outdated, was *CodeCover*. Nevertheless, it is questionable if knowing which test cases intersect is enough information for deciding whether there is a redundancy or not. As for assessing the quality of a test suite, understanding semantic relationships between tests and source code is also a crucial factor for deciding whether tests are redundant or not.

3.3.2 Interpretation

The findings in this chapter clearly emphasize the necessity of further enhancements of classical code coverage analysis and visualizations as proposed in the introduction of this thesis (Chapter 1) and furthermore affirm the relevance of the defined research questions (Section 1.2). The notion of the novel metric *code coverage density*, which describes how test cases are distributed over specific parts of a software, as well as putting more emphasis on how tests and source code are related could fill the encountered gaps concerning information needs in the area of refactorings, test suite quality assessment and the process of finding and analyzing test redundancies.

Metric Definition

In this chapter, the definition of the novel metric *code coverage density* is given. The definition is made strictly according to the basic definition of a software quality metric, which is given in Section 2.2. The definition is provided using a bottom-up approach for the specific granularity levels of a program written in an object-oriented programming language. This means, that the definition is first given for finer granularity levels (e.g., LOCs, methods) and afterwards for coarser levels (e.g., classes, packages).

4.1 Line Coverage Density

The Line Coverage Density (LineCovDens) is a value, that states how many tests of a given test set cover a specific LOC. The set of tests to be considered can either be a predefined subset of the test suite (e.g., which are of interest or form a logical unit) or the test suite in its entirety.

The inputs for the metric are a specific line LOC_j and a set of n tests $T = \{T_1, \dots, T_n\}$. The computed LineCovDens for LOC_j , which is denoted as $LineCovDens_{LOC_j}$, is therefore the number of tests in T that cover LOC_j :

$$LineCovDens_{LOC_j} = \sum_{m=1}^n isCoveredBy(LOC_j, T_m) \quad (4.1)$$

The function *isCoveredBy* is defined as follows:

$$isCoveredBy(LOC_j, T_i) := \begin{cases} 1, & \text{if } LOC_j \text{ is covered by } T_i \\ 0, & \text{if } LOC_j \text{ is not covered by } T_i \end{cases} \quad (4.2)$$

Note that “is covered/not covered” refers to the concept of *statement coverage* as described in Section 2.3.4.

The Relative Line Coverage Density ($\%LineCovDens$), which states how many percent of the tests considered cover the respective line, is defined as follows:

$$\%LineCovDens_{LOC_j} = \frac{LineCovDens_{LOC_j}}{|T|} \times 100 \quad (4.3)$$

Example: Table 4.1 shows the coverage for four LOCs ($LOC_1 - LOC_4$) of a test set $T = \{T_1, \dots, T_4\}$. It holds that two tests (T_2 and T_3) are covering LOC_1 , which is denoted with 1 in the table. The remaining tests do not cover the LOC. Therefore, two out of four tests cover the line, which results in a LineCovDens of 2 and a $\%LineCovDens$ of 50,00%. LOC_2 has a LineCovDens of 4 and a $\%LineCovDens$ of 100,00%, which means that all tests of the given test set cover this line. The values for LOC_3 and LOC_4 are computed analogously.

	T_1	T_2	T_3	T_4	<i>LineCovDens</i>	<i>%LineCovDens</i>
LOC_1	0	1	1	0	2	50,00%
LOC_2	1	1	1	1	4	100,00%
LOC_3	1	0	1	1	3	75,00%
LOC_4	0	1	0	0	1	25,00%

Table 4.1: LineCovDens example

4.2 Method Coverage Density

The Method Coverage Density (MethodCovDens) is defined as the arithmetic mean value of the coverage density values for the lines that form the method. The inputs are therefore a method $M_j = \{LOC_1, \dots, LOC_m\}$ and a set of n tests $T = \{T_1, \dots, T_n\}$. The computation formula is defined as follows:

$$MethodCovDens_{M_j} = \frac{1}{|M_j|} \times \sum_{k=1}^m LineCovDens_{LOC_k} \quad (4.4)$$

The Relative Method Coverage Density ($\%MethodCovDens$) is defined analogous to $\%LineCovDens$:

$$\%MethodCovDens_{M_j} = \frac{MethodCovDens_{M_j}}{|T|} \times 100 \quad (4.5)$$

Note that the reason why the arithmetic mean was chosen for aggregating the LineCovDens on higher abstraction levels is that there are several accompanying statistical values (e.g., minimum, maximum and standard deviation) that support further interpretation on the accuracy of the mean value. Those easily understandable additional values impart a feasible evaluation of the meaningfulness of the coverage densities on higher levels.

Example: Table 4.2 shows the coverage of four LOCs similar to Table 4.1. The depicted lines form the method M_1 , which has a computed MethodCovDens of 2.5 and a %MethodCovDens of 62.50%. This means, that on average 62.50% of the considered test cases cover the method. The computed minimum and maximum values, as well as the standard deviation, state how the concrete values spread around the mean value.

M_1	T_1	T_2	T_3	T_4	<i>LineCovDens</i>	<i>%LineCovDens</i>
LOC_1	0	1	1	0	2	50.00%
LOC_2	1	1	1	1	4	100.00%
LOC_3	1	0	1	1	3	75.00%
LOC_4	0	1	0	0	1	25.00%

<i>MethodCovDens</i> $_{M_1}$	2.5	62.50%
<i>Min</i>	1	25.00%
<i>Max</i>	4	100.00%
<i>Std. Deviation</i>	1.1180	

Table 4.2: MethodCovDens example

4.3 Class Coverage Density

A class C_j can be considered as a set of k methods ($C_j = \{M_1, \dots, M_k\}$). Therefore, for calculating the Class Coverage Density (ClassCovDens) for a given set of n tests $T = \{T_1, \dots, T_n\}$, the MethodCovDens over the union of all methods of that class needs to be calculated. Note that this results in calculating the arithmetic mean value of the coverage density values for the lines that form the class:

$$ClassCovDens_{C_j} = MethodCovDens_{\bigcup_{l=1}^k M_l} \quad (4.6)$$

The corresponding Relative Class Coverage Density (%ClassCovDens) is computed as follows:

$$\%ClassCovDens_{C_j} = \frac{ClassCovDens_{C_j}}{|T|} \times 100 \quad (4.7)$$

Example: Consider a class $C_1 = \{M_1, M_2\}$. The methods of this class are formed by seven LOCs, s.t. $M_1 = \{LOC_1, LOC_2, LOC_3, LOC_4\}$ and $M_2 = \{LOC_5, LOC_6, LOC_7\}$. Given the preliminary definitions of ClassCovDens and MethodCovDens the following equations hold for this example:

$$ClassCovDens_{C_1} = MethodCovDens_{\{M_1, M_2\}} \quad (4.8)$$

$$MethodCovDens_{\{M_1, M_2\}} = \frac{1}{7} \times \sum_{k=1}^7 LineCovDens_{LOC_k} \quad (4.9)$$

Therefore, for computing the `ClassCovDens` and the respective `%ClassCovDens`, the `LineCovDens` values for each LOC needs to be calculated beforehand as depicted in Table 4.3.

M_1	T_1	T_2	T_3	T_4	<i>LineCovDens</i>	<i>%LineCovDens</i>
LOC_1	0	1	1	0	2	50.00%
LOC_2	1	1	1	1	4	100.00%
LOC_3	1	0	1	1	3	75.00%
LOC_4	0	1	0	0	1	25.00%

M_2	T_1	T_2	T_3	T_4	<i>LineCovDens</i>	<i>%LineCovDens</i>
LOC_5	0	0	0	0	0	0.00%
LOC_6	0	0	0	1	1	25.00%
LOC_7	1	0	1	1	3	75.00%

Table 4.3: `ClassCovDens` example: computation of `LineCovDens` values

Given the above values and the equations 4.8 and 4.9, the `ClassCovDens` of C_1 is therefore computed as follows:

$$ClassCovDens_{C_1} = \frac{2 + 4 + 3 + 1 + 0 + 1 + 3}{7} = 2 \quad (4.10)$$

Simplified, computing the `ClassCovDens` for a class results in calculating the mean value of the line coverage density values for all LOCs of all methods that form the respective class. As for the `MethodCovDens`, the minimum and maximum coverage densities, as well as the standard deviation are considered as depicted in Table 4.4.

<i>ClassCovDens_{C_j}</i>	2	50.00%
<i>Min</i>	0	0.00%
<i>Max</i>	4	100.00%
<i>Std. Deviation</i>	1.3093	

Table 4.4: `ClassCovDens` example: values for class C_1

4.4 Higher Granularity Levels

The coverage density calculations on higher granularity levels such as packages and modules can be done in analogy to the computations on class level.

The calculation for a package P_j , which is usually composed of a set of k classes s.t. $P_j = \{C_1, \dots, C_k\}$, would be made through calculating the `ClassCovDens` over the union of all classes that form the package P_j . This leads to computing the `MethodCovDens` over the union of all methods that form those classes.

The same holds for modules and higher granularity levels.

CHAPTER 5 

Concept

This chapter summarizes the activities conducted during the conceptional phase of this thesis (see Section 1.3.2). In Section 5.1, the proposed requirements that have been established based on the outcomes of the research phase (Section 3.3) and the scenarios given in the initial problem description (Section 1.1) are described. Section 5.2 describes the conduction of the expert interviews and the outcomes of those. Finally, Section 5.3 documents needed adaptations of the initially defined requirements and summarizes the final set of requirements that form the basis for the implementation phase (see Section 1.3.3).

5.1 Concept Proposal

This section proposes the basic concept for the planned prototype. This includes, on the one hand, a definition of fundamental requirements on the basis of the identified information needs as stated in Section 3.3. On the other hand, concrete realization proposals, which form the basis for the expert interviews described in Section 5.2, are presented and explained.

5.1.1 General Overview on Test Distribution (*R1*)

The metric *Code Coverage Density* should be visualized for the artifacts of an entire application. This means in detail, that for typical object-oriented programs, the metric's higher abstraction levels (i.e., ClassCovDens and higher levels) should be visualized with respect to the artifacts (i.e., classes, packages, etc.) they correspond to.

Realization Proposal: Figure 5.1 shows a possible visualization approach that satisfies this requirement. The ClassCovDens, as well as higher granularity levels (i.e., on package, module and project level), are visualized in a sunburst diagram, which is created in accordance to the typical structure of object-oriented programs/applications. The inner sections denote modules and packages. The outer sections denote concrete classes. The color gradient visualizes the coverage density of the respective section, where a darker blue tone means a higher emergence of test cases. A brighter tone on the contrary means that there are less covering tests.

Corresponding Information Needs: The target of this requirement is to give an appropriate insight concerning the distribution of the test cases over the entire project and, moreover, to achieve a better understanding concerning the relation between test cases and source code on higher abstraction levels (*IN5*).

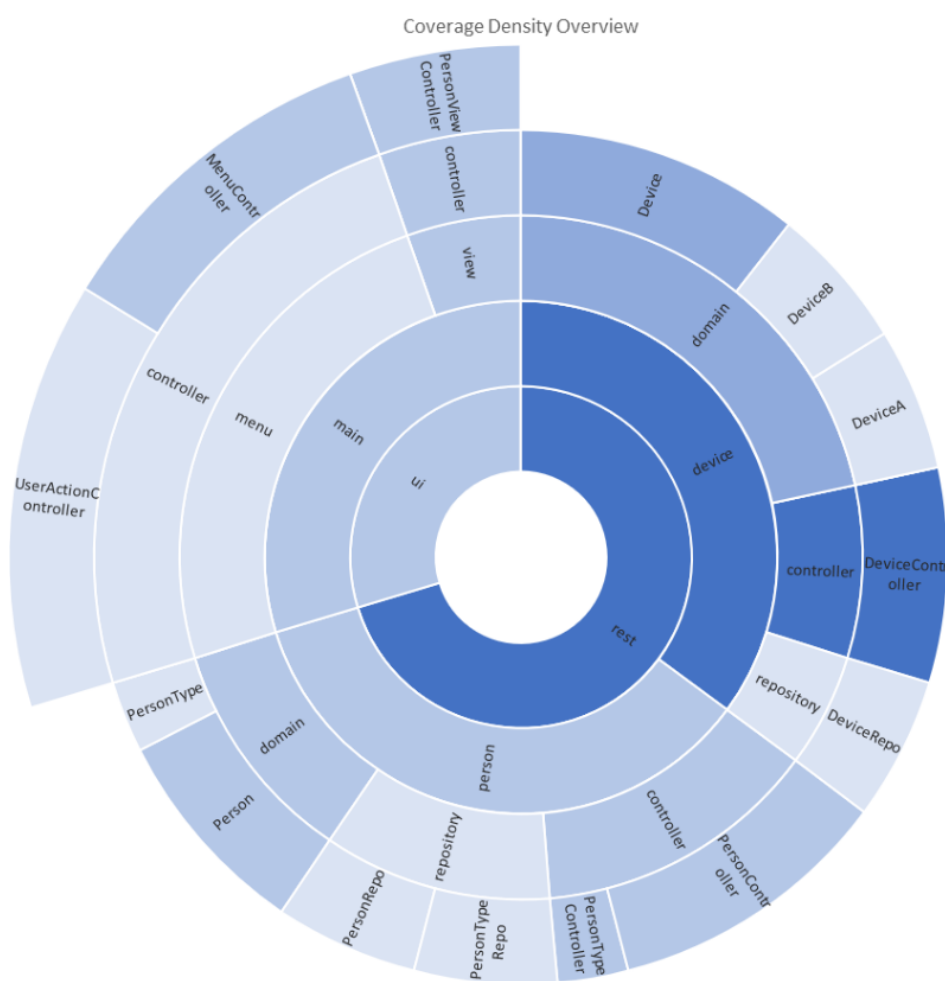


Figure 5.1: Realization Proposals: Sunburst Diagram

5.1.2 Overall Statistics (R2)

The metric's dispersion measures (i.e., the mean, minimum and maximum coverage density, as well as the standard deviation) should be presented together with classical coverage metrics (i.e., overall, statement, method and branch coverage) for the various abstraction levels of an application (i.e., classes, packages, etc.).

Realization Proposal: Considering the previously presented sunburst diagram, it should be possible to open a detail view for each diagram section which shows the needed overall statistics about coverage and coverage density (Figure 5.2).

Corresponding Information Needs: Presenting detailed metric data about a specific artifact is a de-facto standard feature in all available coverage tools and, moreover, necessary and helpful in the context of all identified information needs in Section 3.3.

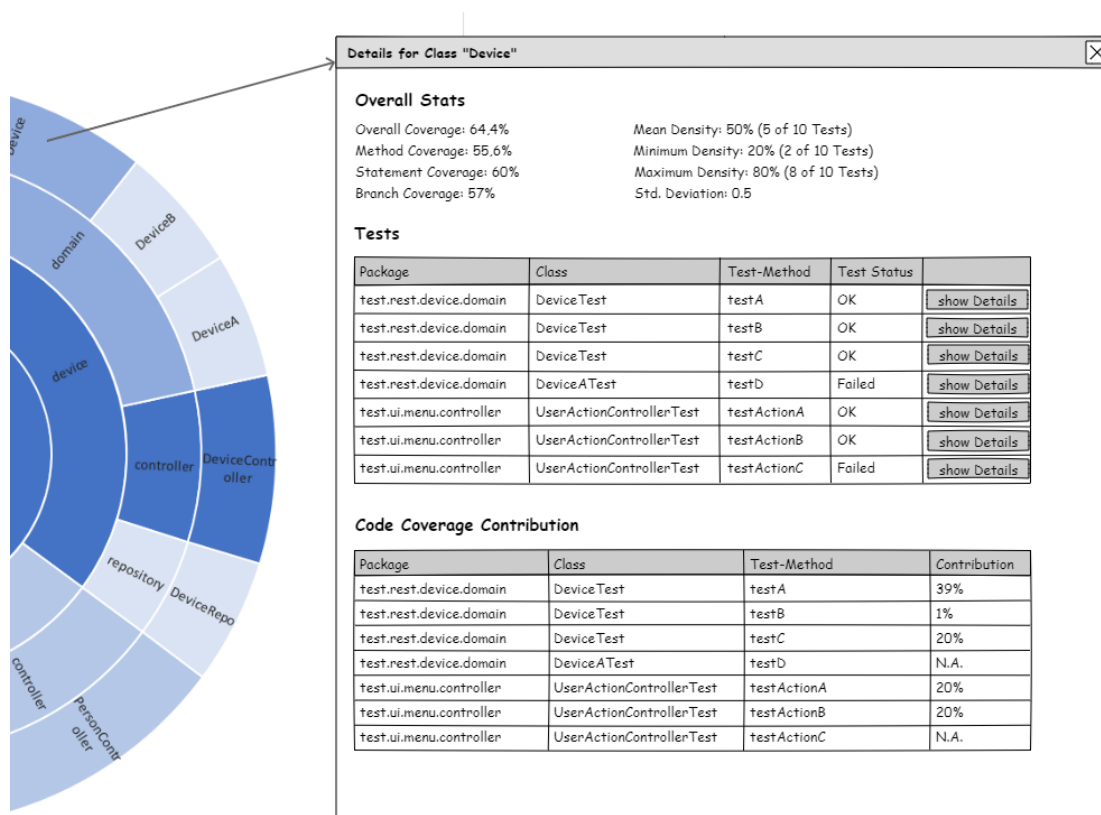


Figure 5.2: Realization Proposals: Sunburst Details

5.1.3 Relation of Test Cases and Higher-Level Artifacts (*R3*)

The relation between source code and tests should be pointed out on higher abstraction levels by stating which test cases cover which artifacts of a program (i.e., classes and packages).

Realization Proposals: As shown in Figure 5.2, the sunburst’s details also show which tests cover the selected section. In addition, there is a listing which shows the coverage contribution of the tests concerning the respective section.

Corresponding Information Needs: The requirement mainly aims at a clearer understanding of the relation between test cases and software artifacts, which could henceforth support in assessing the quality of a test suite (*IN5*, *IN6*).

5.1.4 Test Distribution for a Specific Class (*R4*)

The distribution of test cases should be presented for a specific class with reference to the LOCs it is composed of. More specifically, the LineCovDens should be visualized for all LOCs of the class in order to highlight parts of the code that are covered through more/less tests. Furthermore, it should also be visible which concrete test cases cover which LOC.

Realization Proposal: Figure 5.3 shows a possible class visualization which visualizes the LineCovDens in the context of a Java class *rest.device.domain.Device*. Each line of code gets an assigned number, which states how many tests cover the respective line. The color gradient visualizes these values. A line with a darker blue tone denotes that there are more tests covering it. A brighter tone denotes a lower emergence of test cases. Figure 5.4 furthermore shows an extension of the class view previously described. For each line of code, it is possible to view a detailed listing about the covering tests and their state.

Corresponding Information Needs: The intention for proposing this requirement was to give potential users more information about how test cases spread over the inner structure of a class in detail. This means that, in contrast to existing tools, the visualizations do not only show whether lines are covered or not but also how many test cases hit them and how they distribute across the class. Together with the information about which test cases cover a specific line in the class view details, this could significantly support the process of assessing the risk of refactorings and code changes (*IN4*).



Figure 5.3: Realization Proposals: Class View

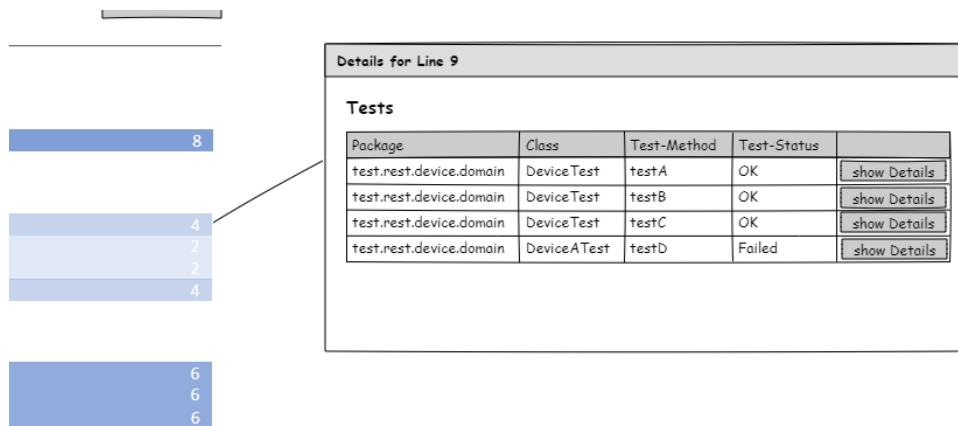


Figure 5.4: Realization Proposals: Class View Details

5.1.5 Test Set Configuration (R5)

In the context of requirement *R4*, it should also be possible to evaluate the distribution of test cases over a specific class with respect to a user-defined test set. This means especially, that the test set considered for computing the LineCovDens should be configurable by the user.

Realization Proposal: Figure 5.5 shows a possible extension of the class view presented in Figure 5.3. Within this dialog, a potential user would be able to select the tests that should be considered for visualizing the LineCovDens within the respective class view.

Corresponding Information Needs: This requirement targets at establishing support in various areas. On the one hand, the feature would provide a possibility to filter tests with specific characteristics (e.g., visualizing the coverage density and covering tests only for unit, integration or system tests), which could furthermore be helpful for assessing the quality of the test suite (*IN5*). On the other hand, filtering test cases in this context could also support during the process of identifying test redundancies, as the visualizations enable a fine-grained analysis of the coverage of specific test cases (*IN7*).

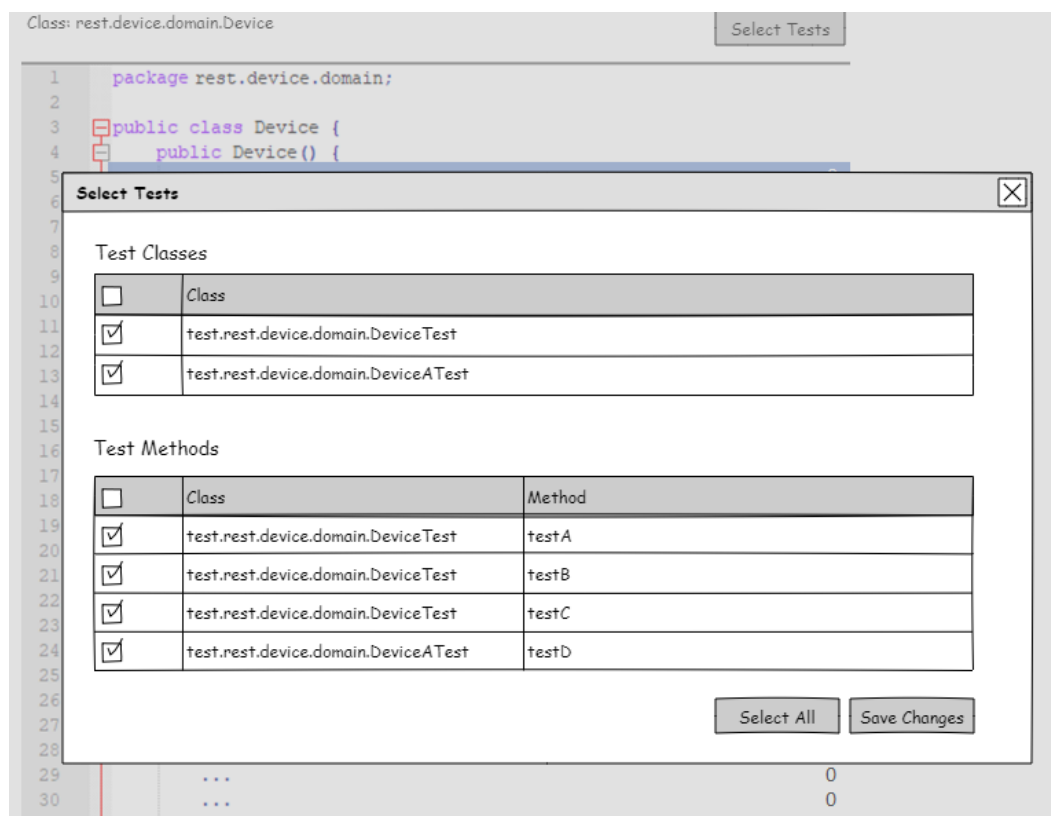


Figure 5.5: Realization Proposals: Test Set Filter

5.1.6 Test Distribution Comparison (*R6*)

It should be possible to compare the computed MethodCovDens, as well as the mean, minimum and maximum coverage density, for a selected set of methods and covering tests.

Realization Proposal: The mockup presented in Figure 5.6 visualizes the distribution of four test cases over the methods $M1 - M5$ in a radar chart. The visualization states the mean coverage density (blue line), as well as the minimum and maximum coverage density (orange and grey lines) for the depicted methods. The visualization is customizable, which means that it is possible to select the considered tests and methods. Alternatively, a comparison of the test distribution could also be achieved by a bar chart similar to the radar chart (Figure 5.7). For each method in the radar/bar chart, it is also possible to open a detail view, which states overall statistics of the method and lists the covering tests (Figure 5.8). Furthermore, there is a listing of all lines of code that form the respective method, together with their computed coverage density values.

Corresponding Information Needs: The intention behind this requirement was to establish further support for assessing the quality of a test suite (*IN6*). With the proposed visualizations, the evenness of the distribution of test cases that cover a set of methods could be analyzed very easily, as a not evenly distribution would, i.e., generate a graph with outlying edges in the radar chart (e.g., method $M4$ in Figure 5.6). On the contrary, it holds that the more evenly the graph spreads, the more evenly the test cases are distributed. The bar chart visualization would show similar behaviors in such situations.

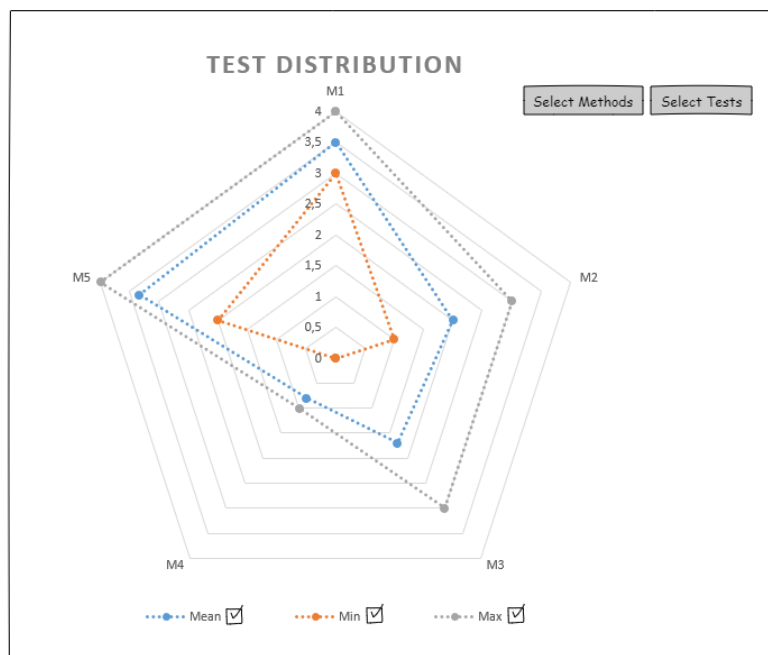


Figure 5.6: Realization Proposals: Radar Chart

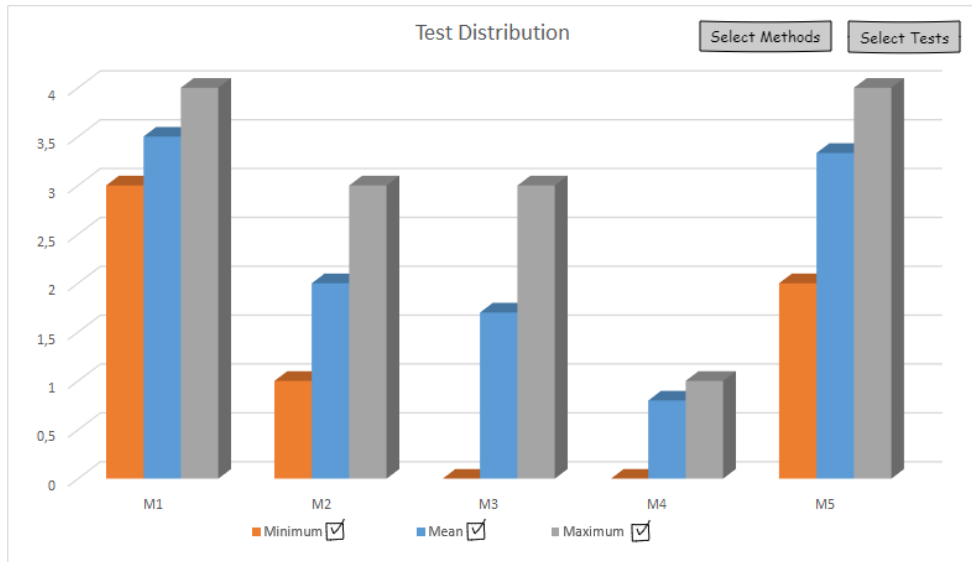


Figure 5.7: Realization Proposals: Bar Chart

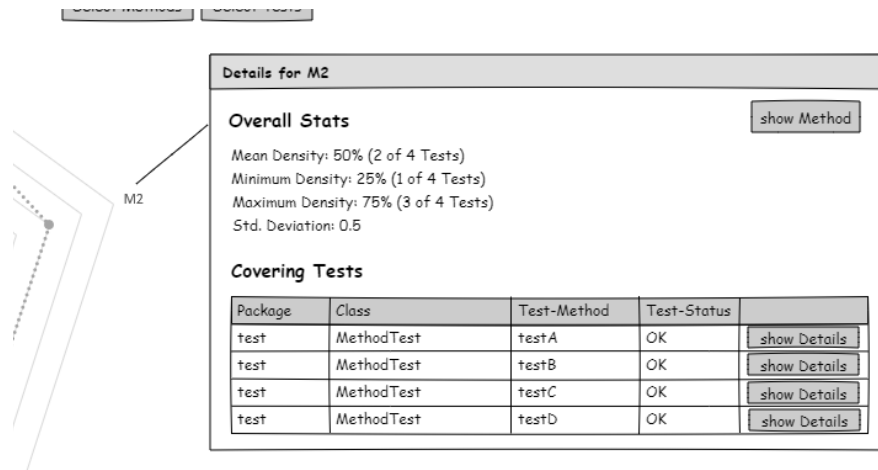


Figure 5.8: Realization Proposals: Radar Chart Details

5.1.7 Test Details (R7)

For each test case, it should be possible to identify the classes covered by the respective test case.

Realization Proposal: The mockup shown in Figure 5.9 shows a listing of classes that are covered by a specific test case. In addition, the listing shows the respective coverage contribution of the test.

Corresponding Information Needs: In combination with all other requirements, this shall in the first place give even denser insights concerning how test code and source code is related in order to support test suite maintenance and the identification of test redundancies (*IN5*, *IN7*).

Test Details

Name: test.nest.device.domain.DeviceIntegrationTest
Status: OK

Covered Classes

Package	Class	Method	Coverage Contribution (Class)	
rest.device.domain	Device	execute()	12%	show Class
rest.device.domain	Device	eval()	9%	show Class
rest.device.domain	Device	destroy	20%	show Class
rest.device.domain	DeviceA	destroy	10%	show Class
ui.main.menu.controller	UserActionController	commit()	5%	show Class
ui.main.menu.controller	UserActionController	evaluateInput()	2%	show Class

Figure 5.9: Visualization Proposals: Test Details

5.2 Expert Interviews

The following sections describe how the conducted expert interviews were planned, executed and evaluated. The concrete intentions of those interviews were as follows:

- Evaluating the metric definition as described earlier in Chapter 4
- Obtaining a relevance ranking for the abstraction levels of the metric
- Presenting the feature and visualization proposals as presented in Section 5.1
- Obtaining a relevance ranking for the presented drafts
- Gathering further ideas and information needs in this context
- Discussing the idea, the metric and the realization proposals

The outcomes of those interview sessions formed the basis for the definition of the final set of functional and non-functional requirements (Section 5.3).

5.2.1 Interview Sessions

The interviews were conducted in three separate online video sessions (one session per participant). At the beginning of each session, the interviewee was first introduced into the overall topic of the thesis and the concern of the interviews. Afterwards, the conductor went through a questionnaire together with the participant, which was created with an online questionnaire tool. The form was filled by the conductor, who shared his screen to the interviewee and went through the questions step by step. The answers were filled following to the answers and instructions of the participant. At the end of each session, there was a short closing discussion.

The interview procedure and the questionnaire itself were furthermore evaluated through a preceding pilot interview. The intention was in the first place to check whether the questions have been formulated in a suitable fashion and were not misleading or lacking of important information in order to get a consistent set of answers. Based on this pilot session, the questionnaire was adopted respectively.

Each session was audio-recorded in order to recapitulate the discussions and remarks that aroused. Note that the recordings were not used for further processing of the data (e.g., transcription), as all the remarks and discussion outcomes that have aroused were instantly logged and written down. The only purpose of the audio records was to be able to rehear the interviews if necessary.

5.2.2 Participants

The participants were three experts from the domain of software engineering. All interviewees were between 25 and 34 years old and had experience in the industrial area. One of the participants had less than five, one between five and ten and one between eleven and 20 years of experience in the industrial sector. Furthermore, two of the participants were also working in the scientific area, whereas one had less than five and the other one between five and ten years of experience.

5.2.3 Questionnaire

The interviews followed a semi-structured approach with both quantitative (closed) and qualitative (open) questions. The questionnaire consisted of two distinct parts. The first one was directed at questions concerning the metric evaluation, while the second one included questions concerning the visual drafts and feature proposals. The following sections describe the questions asked in detail and summarize the overall outcomes. A full list of all questions can be found in the appendices (Section A.1).

Metric Evaluation

During the first part of the interview, the participants received an overall introduction to the metric calculation. Specifically, the conductor explained each level of abstraction of the metric (LineCovDens, MethodCovDens, ClassCovDens and higher) and presented the

calculation steps through a speaking example, whereby each level got a separate question. In addition, the possible dispersion measures were presented. For each of the examples, the interviewees were asked how relevant this information is to them and whether they had additional remarks on the shown concept. For the relevance rating, a 5-point scale was used which reached from 1 = *not relevant* to 5 = *highly relevant*. Those questions are shown in the appendix in Figure A.2 and A.3. Furthermore, the participants were asked to give a ranking on how important the metric on each level of abstraction (LOC, method, class and package level) is in their opinion (Figure A.4). Finally, they were enquired whether the metric is clearly defined and understandable and if they had any further remarks on the presented concept.

Mockup Presentation and Evaluation

In the second part, the realization proposals as documented in Section 5.1 were presented to the participants. For each of the presented visualizations, the conductor explained what the concrete idea and intention behind the feature is, as well as how it is connected to the metric calculation presented earlier. For each of the mockups, the participants were asked to give a relevance rating on how important such a feature is in their opinion (again on a 5-point scale) and whether they have additional remarks or further ideas. Summarizing, the target of this procedure was mainly to evaluate these first visualization ideas, extend or discard them if necessary, and perceive a requirement/feature ranking based on the participants importance ranking.

5.2.4 Outcomes

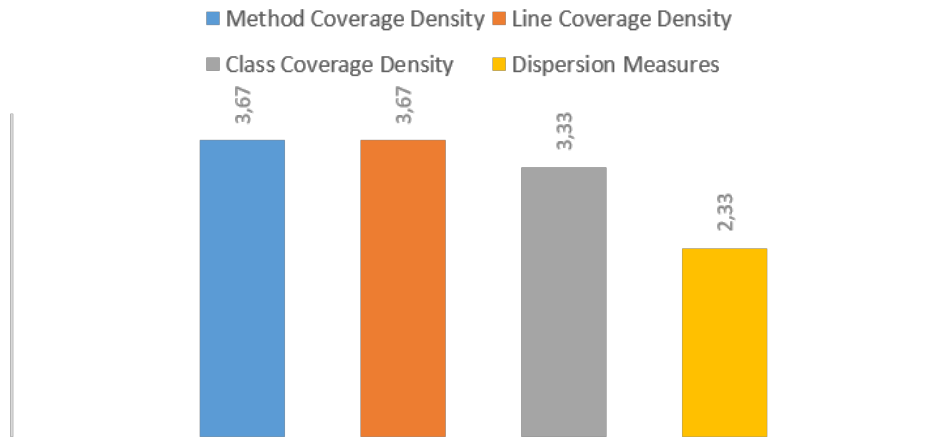
The evaluation of the questionnaire was conducted as follows. For each quantitative question (i.e., all relevance ratings), the mean value of all given ratings was considered for further analysis. For each qualitative question (i.e., further ideas and remarks, as well as the closing discussion), the statements given by the participants were logged as plain text. This additional information will just be stated and summarized in the following.

Metric Relevance

Concerning the metric, all participants stated that it is clearly defined and understandable in their opinion. Regarding the introductory explanation of the coverage density calculations, MethodCovDens and LineCovDens got the highest relevance ratings, followed by ClassCovDens and the dispersion measures (Figure 5.10a).

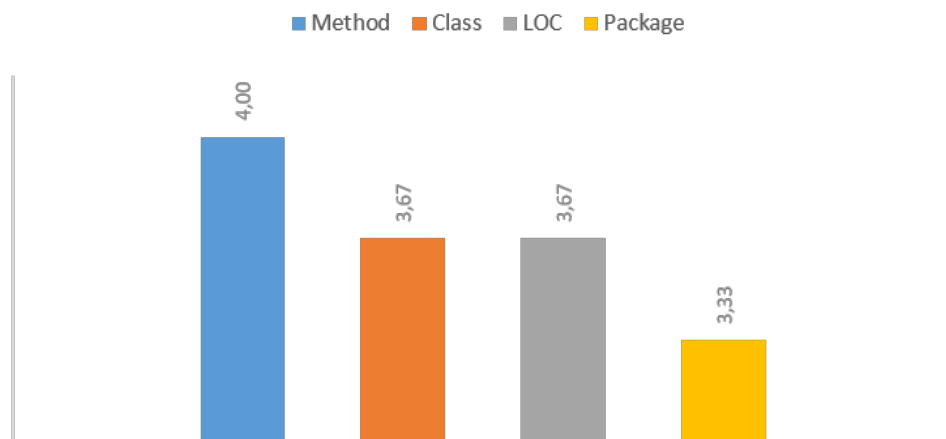
Concerning the relevance of coverage density on the different abstraction levels, the calculation on method level was rated as most important (Figure 5.10b). The relevance on class and line level was equally rated, whereas the relevance on package level was rated as less important.

MEAN METRIC RELEVANCE



(a) Relevance rating of coverage density calculation

MEAN LEVEL RELEVANCE



(b) Relevance rating of coverage density on different abstraction levels

Figure 5.10: Metric relevance ratings (1 = *not relevant*, 5 = *highly relevant*)

All three participants stated that they had problems to imagine what a good or bad value of the indicator is. When they were introduced to LineCovDens, they remarked that they are missing a baseline or a reference value to decide whether the actual value computed can be seen as desired or undesired. One participant noted that making decisions or conducting further steps based on the metric may also depend highly on the context of the considered LOC, method or class. For example, if the LOC considered is a log statement, it may be ok if the coverage density is low. On the other hand, if the LOC is,

e.g., part of a loop (or “actual” code in general) a low density may indicate that further testing is needed. Furthermore, another participant noted that a high density may also imply that there are test duplicates. The latter is a train of thought that was nevertheless a base idea of the metric and the visualizations.

Though the dispersion values were rated as the least important indicators in the questionnaire, two participants stated that the minimum and maximum coverage density may be useful to a certain extent, e.g., for finding out that there are lines that are covered very often, but also those which are not covered at all. Both remarked, that they would not use the standard deviation for any further analysis or make decisions based on this value.

Concerning the importance of the metric on the various levels of granularity (LOC, method, class, package, etc.), there was one participant explicitly stating that in his opinion, the relevance mainly depends on the use case that drives a user to use the metric. The coverage density on the level of LOCs may be interesting if the test suite quality is already quite high and further improvements are pursued. But if a user wants to get an initial overview in a project, where test suite quality may not have been assessed until now and first actions need to be taken, higher abstraction levels may be of higher interest.

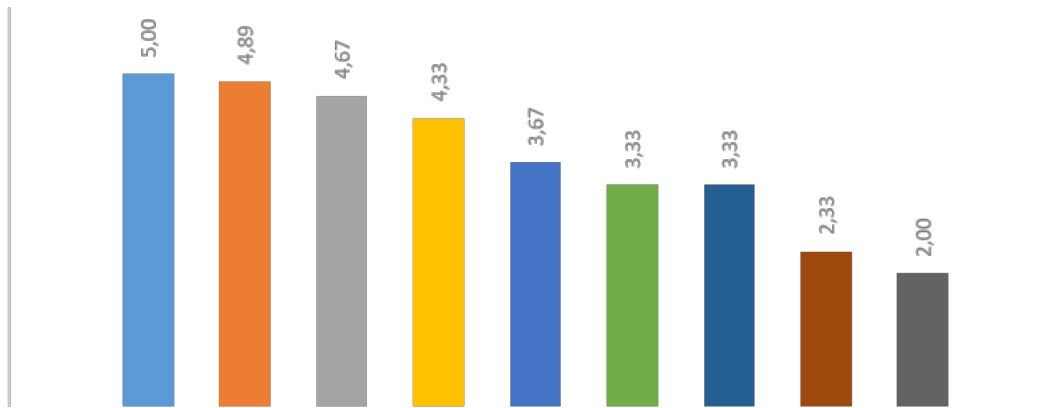
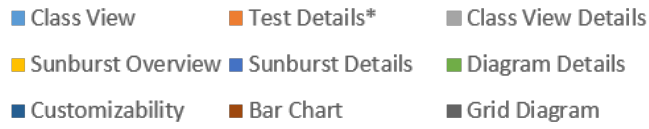
Feature Relevance

The mean ratings for the proposed features of the visualization prototype are depicted in Figure 5.11a. Note that for the feature *Test Details*, there was no single relevance rating in the questionnaire and therefore the mean value of the mean rating for the questions about the relevance of the detail view for different test levels was used. The features *Class View*, *Test Details* and *Sunburst Diagram* were rated as most appealing (together with their detail views), whereas *Radar Chart* and *Bar Chart* were considered as not relevant to the participants. Concerning the relevance of the test details visualization, the participants found that it is of high importance if the test considered is a unit test or an integration test but less important if the test is a system test (Figure 5.11b).

Similar to the remarks concerning the metric definition, one participant explicitly noted that it is hard to decide whether a dark/light blue tone in the diagrams is something that can be considered as good/bad and that this is highly context dependent. Regarding the sunburst diagram, one participant stated that the usefulness of this visualization also depends on the customizability and the data in scope. The interviewee explicitly remarked that it would be useful to tell the tool “how” the sunburst should behave.

For certain use cases, it may not be wishful to see the whole architecture of the software and sections should be expendable one after the other. For other use cases, it may be necessary to get an overview over the whole architecture and all sections should be expanded per default. Furthermore, a participant remarked that the usefulness is absolutely dependent on the user’s domain knowledge.

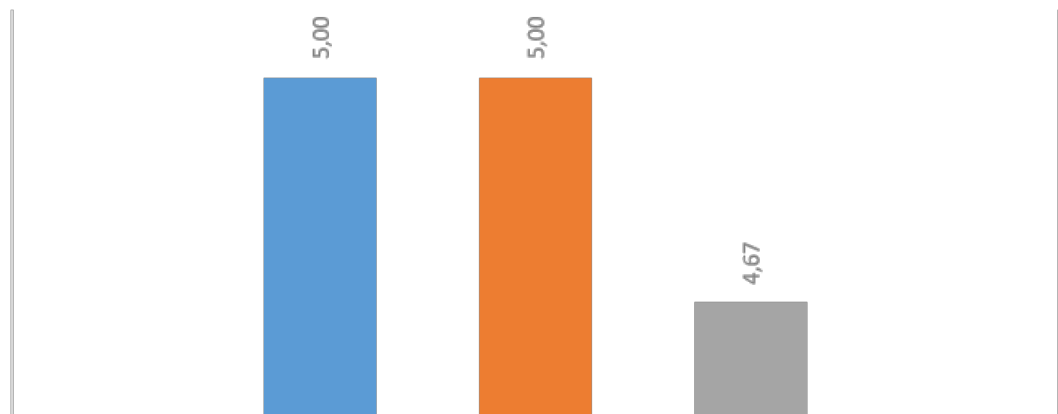
MEAN FEATURE RELEVANCE



* mean value of the mean ratings concerning Test Details for a unit, integration or system test

(a) Relevance rating of proposed features/mockups

TEST DETAILS TYPE RELEVANCE



(b) Relevance rating of the test details mockup for different test levels

Figure 5.11: Feature/Mockup relevance ratings (1 = not relevant, 5 = highly relevant)

Concerning the detail views, one participant asked why failed tests appear and what the intention behind this information is. It should be overthought whether those tests need to be included and what the advantages/disadvantages could be. Another participant stated that showing details (and implicitly the data that drives the indicators) is always good and wishful. Hence, in the opinion of the interviewee, such detailed information may not be used very often in industrial projects due to lack of time and resources.

The radar chart and the bar chart were least appealing to the interviewees. Though, two participants stated that (if it was available) they would rather prefer the bar chart over the radar chart, as it seems more intuitive. Furthermore, one participant remarked that the radar chart may be tempting towards comparing methods that are not intended to be compared. The bar chart visually separates the methods from each other, which is considered to be a better way of presentation as in the opinion of the interviewee, methods should not be compared based on their coverage density. In addition, one participant stated that he would not use the radar chart or the bar chart at all and that the class view would be the most preferred visualization.

Regarding the test details, one participant remarked that the coverage contribution value is confusing in this visualization. The interviewee stated that this metric would make more sense in the context of tests that cover a specific class, but not for classes that are covered by a specific test.

General Discussions

During the closing discussions, there were various additional and interesting statements that needed to be captured. One participant explicitly stated that in his opinion (as it is the case for many software metrics), it may only be reasonable to make decisions based on coverage density in combination with other metrics (e.g., code coverage, code complexity, etc.). Regarding maintenance and refactoring tasks, one participant stated that MethodCovDens and the visualizations based on this indicator may be the most important ones, as those tasks usually deal with this level of abstraction.

Summarizing, the participants stated that they think that the following factors highly influence the usefulness and relevance of the metric and the visualizations:

- What is the user's intention?
- What is an adequate value to aim for? Is there a possibility to generate recommendations?
- In order to give reasonable statements about the artifacts under review, high domain knowledge is needed.
- Who is the target audience? The relevance may be different for each interest group (software developers, testers, managers, etc.).

In addition, the following enhancing ideas and features were proposed during the closing discussions:

- Integration in other tools like common IDEs, SonarQube etc.
- A ranking list of classes/tests due to their coverage density they have/contribute
- An intersection visualization where classes and test cases can be compared concerning the covered LOCs, which would help to find possible intersections between tests

5.2.5 Summary

At a glance, the following drafts and proposals were most appealing to the experts and, therefore, important candidates for a feature set of the implemented prototype. The list is already sorted by the importance of the feature:

1. Class view (with respective detail view)
2. Test details
3. Sunburst diagram (with respective detail view)

Both the radar chart, as well as the bar chart will be neglected due to their low relevance ratings and the qualitative feedback of the questionnaires.

Furthermore, the following takeaways were captured for the next phase of the thesis:

- All visualizations must be highly customizable.
- Further graphical user interface (GUI) design in the requirement analysis and definition phase should consider the feedback stated in this section.
- The dispersion measures had low relevance ratings, hence the minimum and maximum coverage density values could be of use in certain cases.
- The factors that influence the usefulness of the metric and the enhancing ideas for further features should be kept in mind in further phases - as summarized earlier based on the general discussions.

5.3 Adaptions

In this section, necessary adaptions of the initially defined functional requirements and realization proposals presented in Section 5.1 that were made based on the outcomes of the expert interviews (Section 5.2) are documented. Furthermore, the non-functional requirements are defined. Last but not least, a summarizing list of the final requirements for the planned prototype is given.

5.3.1 Additional Requirements

Ignoring Failed Tests (*R8*): It should be customizable, whether failed tests are taken into account for the coverage density computations or not.

Ranking of Tests (*R9*): There should be a list of tests that are ranked due to the coverage density they contribute (i.e., how many LOCs they cover). The list should be sorted in descending order, s.t. tests that cover the most LOCs are listed first.

Ranking of Classes (*R10*): There should be a list of classes that are ranked due to the coverage density they exhibit. The list should be sorted in descending order, s.t. classes that have a high coverage density are listed first.

5.3.2 Realization Proposal Enhancements

Customizability of the Sunburst Diagram: With respect to the realization proposal in Subsection 5.1.1, users must be able to configure how the presentation of the test distribution over the project behaves. For the sunburst visualization, this means in particular that per default, only the first three sections should be visible. If a user clicks on a section, the next outermost section should be expanded. Alternatively, the user should be able to configure an immediate rendering of all sections of the diagram.

Base Coloring: The base coloring (default = blue) of the visualizations should be customizable.

Test Details: The test code (simple code view) should be available in the test details to immediately evaluate what the test executes. The code view should be expandable and hidden by default. Tests that failed should be highlighted in red, all passed tests should be highlighted in green (Figure 5.12).

5.3.3 Non-Functional Requirements

Reusability of Underlying Data (*N1*): The underlying data (e.g., coverage data) should be easily reusable, s.t. the implementation of a visualization in common IDEs or other tools is possible

Integration in other Tools (*N2*): The generated visualizations should be implemented in such a way that they could easily be integrated into CI-Tools (e.g., Jenkins)

Test Details

Name: test.rest.device.domain.DeviceIntegrationTest
Status: Not ok

Covered Classes

Package	Class	Method	
rest.device.domain	Device	execute()	show Class
rest.device.domain	Device	eval()	show Class
rest.device.domain	Device	destroy	show Class
rest.device.domain	DeviceA	destroy	show Class
ui.main.menu.controller	UserActionController	commit()	show Class
ui.main.menu.controller	UserActionController	evaluateInput()	show Class

Test Code [hide](#)

```

1  package rest.device.domain;
2
3  public class DeviceIntegrationTest {
4
5      @BeforeEach
6      public void init() {
7          ...
8          ...
9      }
10
11     @Test
12     public void testA() {
13         ...
14     }
15
16     @Test
17     public void testB() {
18         ...
19         ...
20         ...
21         ...
22     }
23

```

Figure 5.12: Test Details with Test Code

5.3.4 Neglected Realization Proposals

In particular, two of the notions presented in the realization proposals and captured in the expert interviews were omitted in the context of this thesis.

First, displaying the coverage contribution as proposed for the requirements *R3* and *R7* (Figures 5.2 and 5.9) was an idea that arose from the analysis of existing code coverage tools (esp. *OpenClover*), as they also display such values. Hence, this value is very vague in the existing tools and also not scientifically based. Therefore, it may be unclear to the user what the actual meaning is, whereby it was finally neglected in the context of this thesis.

Second, visualizing similar tests as proposed by some participants during the expert interviews (see Section 5.2.4) was also neglected as it would have been necessary to do further research whether tests can be considered as similar based on the percentage of covered lines. This was clearly out of the scope of this thesis, whereby the feature had to be neglected.

5.4 Final Requirements

The final set of functional (F) and non-functional (NF) requirements that form the basis for the following phases of this thesis are summarized and depicted in Table 5.1. In addition, all requirements were classified as either *must-have requirements* (MH) or *nice-to-have requirements* (NTH). Note that requirements denoted as NTH were decided to be marked as such as they are common additional features, that would not essentially enhance the prototype in order to satisfy the identified information needs captured in Section 3.3. Nevertheless, those requirements are documented in this section for future work.

ID	Requirement	Classification	Type
R1	General Overview on Test Distribution	MH	F
R2	Overall Statistics	MH	F
R3	Relation between Test Cases and Higher-Level Artifacts	MH	F
R4	Test Distribution for a Specific Class	MH	F
R5	Test Set Configuration	MH	F
R7	Test Details	MH	F
R8	Ignoring Failed Tests	NTH	F
R9	Ranking of Tests	NTH	F
R10	Ranking of Classes	NTH	F
N1	Reusability of Underlying Data	MH	NF
N2	Integration in other Tools	MH	NF

Table 5.1: Final Requirements



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Implementation

In this chapter, the implementation phase as implied in Section 1.3.3 is described. Section 6.1 gives fundamental information concerning the execution workflow of the planned prototype. Section 6.2 depicts the technical concept in detail. Finally, the resulting reports and visualizations generated by the prototype are discussed in Section 6.3.

6.1 Execution Phases

The basic idea of the implemented prototype was to gather coverage data from existing code coverage tools, rearrange the gathered data, calculate the metric *code coverage density* on various levels and present the outcomes in a human-readable report (with reference to the derived requirements in Section 5.4). The report itself should be a static, self-contained HTML report, which does not need any connections to foreign systems (e.g., databases). For the scope of this thesis, the decision was to provide a working version of the prototype which supports Java projects tested with the JUnit framework.

Beforehand, it was necessary to define which information was required for the metric calculation and which tools could provide those. In fact, the following information was needed:

- **Project Structure:** How is the project structured, i.e., what is the package and class structure?
- **Test Results:** Which test cases have been executed and what was their execution status, i.e., did they pass or fail?
- **Code Coverage:** What are the general coverage metrics for packages and classes, i.e., statement coverage, method coverage, branch coverage and overall coverage?

- **Relation between Tests and Code:** Which test cases cover which LOCs (and therefore, which methods, classes and packages)?

The most promising existing tool for gathering this information was *OpenClover* (see Section 3.2.4), as it has the most evolved and structured output data of all analyzed coverage tools during the thesis. Coverage data can not only be visualized in the form of PDF or HTML reports but may also be output into reusable XML files, which contain the recorded coverage data for the whole project structure in machine readable format. Furthermore, the tool generates a reusable database during code instrumentation and test execution. Specifically, the database contains structured information on which tests cover which LOC. Summarized, the XML files together with the database form the perfect basis for answering the questions above.

After deciding what the base coverage tool for the planned prototype was, it was necessary to define how the rearrangement of the given data should work and how to pass it to the report. Figure 6.1 shows the overall execution phases of the prototype.

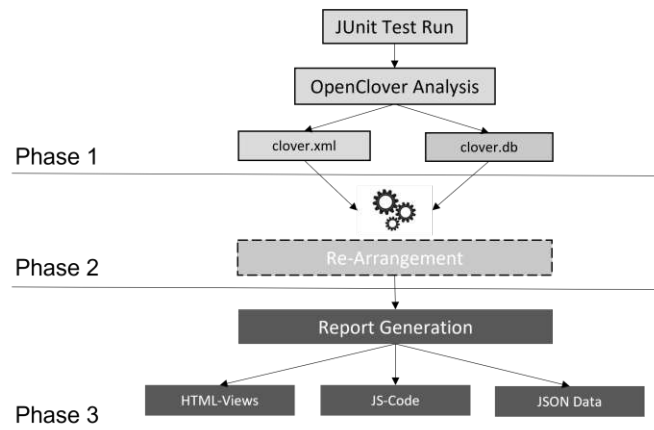


Figure 6.1: Prototype Execution Phases

Phase 1 consists of the execution of the unit tests for the target project and coverage analysis with *OpenClover*. The outputs are the XML files mentioned earlier (clover.xml), together with the generated database (clover.db).

In phase 2, this data gets pre-processed and rearranged. As it was planned to implement an independent report which therefore must also include its underlying data source, the idea was to generate a JSON file, which follows a predefined structure and contains the generated relevant data on the basis of *OpenClover's* output sources. Note that this JSON array does not include the concrete metric values, as the calculation itself depends, e.g., on the test set to be considered, which is configurable by the user (see Section 5.1.5). As calculating the coverage density values on each level for each possible test set in a project would have formed a tremendous overhead, the decision was to do the calculation on the fly within the report, i.e., with client-side business logic.

The final phase 3 takes the processed data and generates the necessary files for the coverage density report. Concerning the report itself, the choice was to generate static HTML files for each view in the report and deliver the business logic in separate JavaScript files, which get included by all the templates. The JSON array should be delivered through a single JavaScript file, which can be accessed through the business logic.

6.2 Technical Concept

Implementation activities were in fact only necessary in the context of phases 2 and 3, as phase 1 only consists of running JUnit tests of the target project and *OpenClover's* coverage analysis in order to get the necessary outputs. The actual implemented program then takes the XML files and *OpenClover's* database as inputs for further processing. The following subsections will describe the technologies used for the latter, as well as the data structure for the JSON file and the directory structure of the final HTML report. Nevertheless, Subsection 6.2.4 will also give an overview on how the overall process (including phase 1) was automated, s.t. the user does not need to execute JUnit tests and *OpenClover* analysis manually in order to run the prototype.

6.2.1 Technologies

The concrete program is a command line tool implemented in Java. Besides the built-in features like classes for reading XML files and loading them into a Document Object Model (DOM), the following third-party libraries were used:

- *OpenClover Plugin*¹, which provides functionality for accessing and reading the coverage database
- *FasterXML/jackson-databind*², which provides Java object serialization to JSON objects and documents
- *Apache Velocity*³, which provides a template engine for generating the report files

Concerning the front-end (i.e., the report files), the front-end development framework *Bootstrap*⁴ was used. Furthermore, the report relies on the following JavaScript libraries and tools:

- *JQuery*⁵, which provides functionality for DOM navigation and manipulation
- *HighlightJS*⁶, which is used for syntax highlighting of the displayed code

¹<https://openclover.org/downloads>

²<https://github.com/FasterXML/jackson-databind>

³<https://velocity.apache.org/>

⁴<https://getbootstrap.com/>

⁵<https://jquery.com/>

⁶<https://highlightjs.org/>

- *vasturiano/sunburst-chart*⁷, which is used for generating the sunburst charts

6.2.2 Data Structure

In the following, the data structure of the JSON array which is generated by the implemented program will be described. A concrete example of a possible data file can be found in the appendix (Section A.3).

The top level of the JSON array describes the project in general. The data structure is therefore composed of the following properties:

- *projectName*: the name of the project
- *coverageMetrics*: the coverage metrics calculated by *OpenClover* (overall, method, statement and branch coverage)
- *packages*: the contained packages
- *testCases*: a listing of the implemented test cases, along with their class names, package names, execution states, file paths and a flag for stating whether the test case is abstract or not
- *numTestCases*: the total number of test cases
- *coveringTests*: a listing of all tests that cover the project, given by their qualified names
- *numCoveringTests*: the number of test cases that cover the project
- *size*: the size of the project, which is the total count of line numbers

An example of this structure is given in Listing A.1. Note that in this snippet, the property “packages” is collapsed and in succession contains all packages of the project along with their sub packages, classes, methods and LOCs. In particular, packages are composed of the following properties:

- *pacakgeName*: the name of the package
- *qualifiedName*: the fully qualified name of the package⁸
- *classes*: the classes contained in the package
- *packages*: the sub packages of the package

⁷<https://github.com/vasturiano/sunburst-chart>

⁸denoted in java package syntax, e.g., “at.tuwien.ac.at.main” for a package “main”

Classes are defined similar to packages along with their contained methods (property *methods*). Methods are composed of LOCs (property *lines*), which are identified by properties for the respective line number and line spans (for statements covering more than one line). In analogy to the project structure, each sub element has the two properties *coveringTests* and *numCoveringTests*. Packages, classes and methods also have properties for their fully qualified name and the computed coverage metrics (analogous to the properties used for the project structure). An example of such a detailed structure is given in Listing A.2.

Note that especially the properties *coveringTests* and *numCoveringTests* are deliberately redundant in order to minimize the computing effort in the business logic. On higher levels (e.g., packages), those properties simply state the union of covering tests of their lower-level elements (e.g., classes, methods and LOCs).

6.2.3 Report Directory Structure

The generated report itself consists of the file *index.html* (which is the actual entry point) and five sub-directories. Figure 6.2 depicts the structure for an example project.

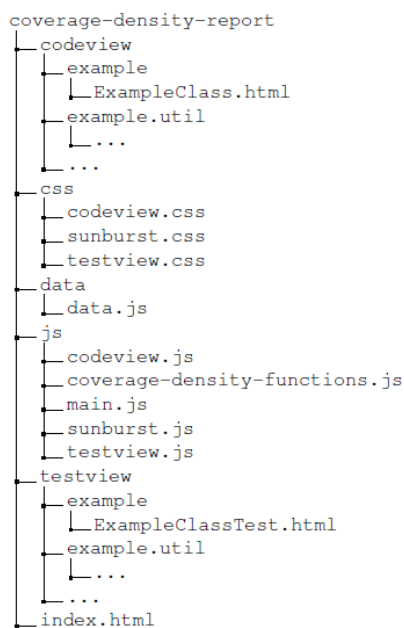


Figure 6.2: Example report directory structure

The directory *data* includes a single file *data.js* containing the JSON data structure described earlier in Subsection 6.2.2.

The folder *js* contains JavaScript files with the business logic for the sunburst, code and test views (e.g., for manipulating the DOM). Furthermore, the file *coverage-density-functions.js* contains all the necessary logic for computing the coverage density metrics on various

levels (based on the underlying JSON data). In detail, the actual metric computation steps as defined in Section 4 are conducted only in this file through independent JavaScript functions.

The folders *codeview* and *testview* contain sub-folders for every single package (identified by their fully qualified names) and furthermore HTML files for every possible view of a class or unit test.

The folder *css* contains the CSS definitions for the respective views.

6.2.4 Execution Automation

As described so far, the common method for generating the coverage density report for a given project is executing the JUnit tests, running *OpenClover's* coverage analysis and feeding the implemented program with the output files as well as the database. For Java projects built with Maven, this would mean that a potential user has to carry out the following steps manually before running the report generator:

1. Include the Maven plugin for *OpenClover* in the projects *pom.xml*
2. Run the unit tests and coverage analysis via Maven
3. Execute the report generator with the given output artifacts

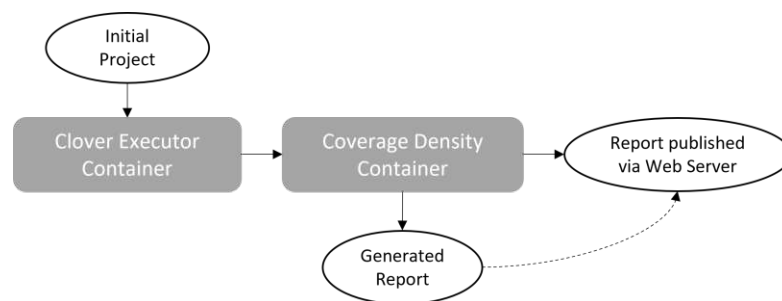


Figure 6.3: Prototype Execution Automation

Especially the step of altering the projects *pom.xml* is not reasonable at all, as an execution of the prototype should not be dependent on changes within the project to analyze. Furthermore, the regeneration of the report for a given project is cumbersome and needs a manual repetition of the above steps. As analytical tools should in the first place be easy to use [48], the idea was to automate the whole process with Docker containers. In fact, exactly two containers are provided for this purpose (see Figure 6.3). Both were built on the basis of the official *OpenJDK* containers⁹.

⁹https://hub.docker.com/_/openjdk

Clover Executor Container

This container executes the unit tests along with the coverage analysis for a given input project. In detail this is done via the execution of Maven and the *OpenClover* plugin via the command line. The plugin itself does not need to be included in the initial projects pom.xml, as it can also be called from the command line externally. To achieve this, the container executes the shell script depicted in Listing 6.1. In order to signal that the container finished with the execution, a file *cd_sync* is written to the target folder of the input project. This file gets removed on line 3 and created on line 7. The concrete Maven execution is triggered on line 6.

```

1 #!/bin/bash
2
3 rm -f /input-project/target/cd_sync
4
5 cd /input-project
6 mvn clean org.openclover:clover-maven-plugin:4.4.1:setup test org.openclover:clover-
  maven-plugin:4.4.1:aggregate org.openclover:clover-maven-plugin:4.4.1:clover -
  Dmaven.clover.generateXml=true -Dmaven.clover.generateHtml=false -Dmaven.test.
  failure.ignore
7 echo "ready" > /input-project/target/cd_sync

```

Listing 6.1: clover-executor.sh

Coverage Density Container

This container waits until the file *cd_sync* is available and afterwards executes the report generator. Furthermore, it starts an *Apache* web server and publishes the generated report, which is in the end available for the user on *http://localhost:8080*.

By following this approach, the (re-)execution of the prototype for a given project is completely automated and does not need any manual intervention of the user.

6.3 Results

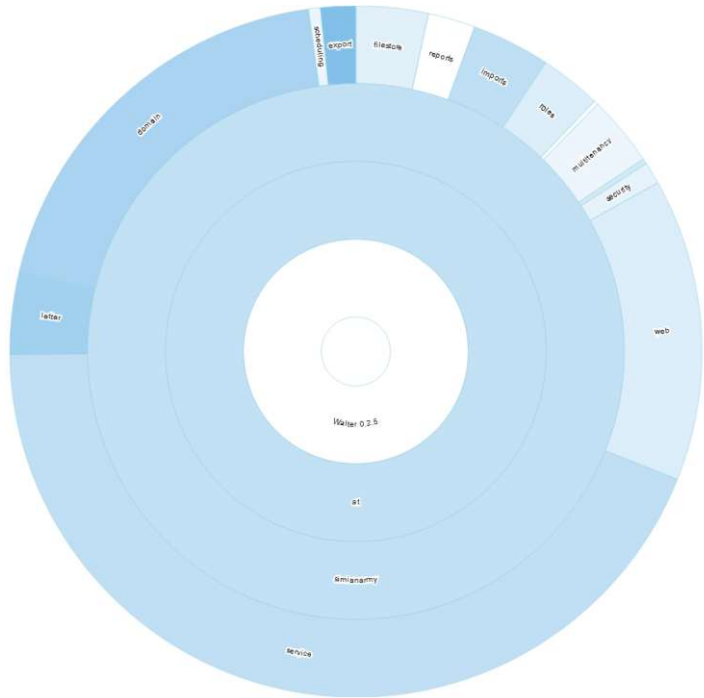
In the following sections, the different views available in the final coverage density report for an example project (which was also the basis for the evaluation in Chapter 7) are described. Note that this is not a detailed hand-book for the usage of the prototype. The focus is, on the one hand, on showing how the requirements in Section 5.4 were implemented with respect to the realization proposals enhancements (Section 5.3.2) and, on the other hand, on stating which additional features had to be implemented due to perceptions that arose during the actual implementation phase.

6.3.1 Sunburst View

The Figures 6.4a and 6.4b show the final version of the sunburst diagram within the generated report, which fundamentally covers requirement *R1*.



(a) with configuration *Show All*



(b) with configuration *Focusable*

Figure 6.4: Coverage Density Report: Sunburst Diagrams

Sunburst Configuration

Figure 6.5 shows how the customization of the sunburst view was realized. It is possible to choose two *Sunburst Behaviors* with the given dropdown input:

- *Focusable*, which means that only the first three layers of the sunburst diagram are shown
- *Show all Levels*, which means that the whole sunburst diagram is rendered immediately



Figure 6.5: Coverage Density Report: Sunburst Diagram Configurations

In addition, the behavior configuration has been expanded with two additional range controls. Those configurations were not defined in the initial requirements and were added during the implementation process as several visualization problems aroused with the example project. In the following, the two range controls, along with the justification why they had to be added, will be described.

Scale Factor

This control allows the user to scale the diagram, which is especially helpful in behavior mode *Show all Levels*. If the diagram has a specific size, not all sections can be labelled with their respective package and class names due to the lack of space (see also Figure 6.4a). Therefore, a possibility was needed to upscale the diagram in order to receive more space within the sections and label them where possible.

Coloring Factor

The control *Coloring Factor* intensifies the coloring of the sections. This means, that the coloring factor (which is calculated on the basis of the coverage density for the respective section) gets multiplied by the configured factor. In detail, this has two very important effects for the diagram. At the one hand, it holds that for large projects with many test cases, the coverage density values may become very low for specific sections. Imagine a project with hundreds of test cases and a class that only gets touched by a few of them. This would lead to a very low relative coverage density, which implies a very light (and often hardly visible) color tone. This may give the impression, that there are no tests covering this section. On the other hand, it may be very difficult to differentiate between sections, that have similar relative coverage densities which result in a nearly equal color tone. The effect is, that those different color tones may not be distinguishable for the user. Scaling the coloring factor is therefore inevitable in order to gain as much information as possible with the sunburst diagram.

Detail View

In order to fulfill the requirements *R2* and *R3*, a detail view was implemented as shown in Figure 6.6. The view appears as the user clicks on a section in the diagram. This requirement has also been slightly adapted for the behavior mode *Focusable*, where a click on a section should also zoom into the clicked section. This would have resulted in a contradiction when this mode is configured, as one user interaction would have triggered two events. Therefore, a mouse click triggers a context menu which enables the user to choose whether to focus on the respective section or to show the details dialog. Note that the info buttons in the list of covering tests lead to the respective test view, whereas the info button in the right corner of the view leads to the class view. The status of each covering test has been visualized through green (passed) and red (failed) thumbs up signs.

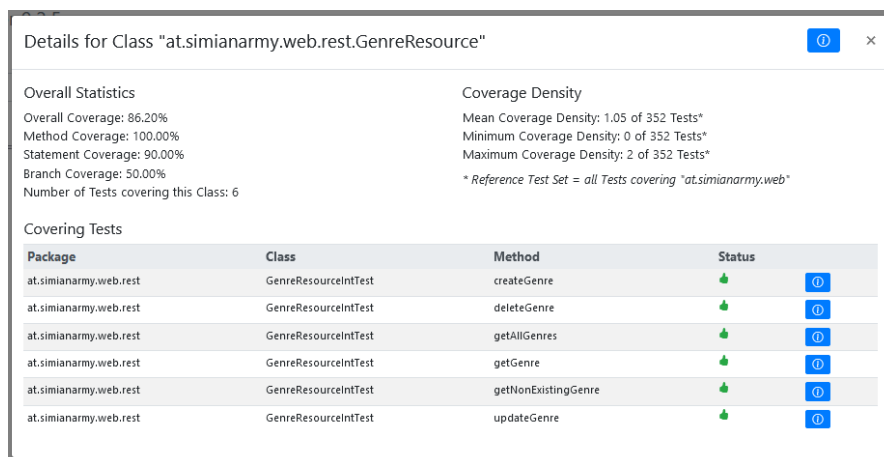


Figure 6.6: Coverage Density Report: Sunburst View Details

Further Adaptions

Furthermore, there were marginal adaptions and additions concerning the standard reference test set for coverage density calculations within the diagram. In earlier versions of the prototype, the reference test set for the calculations (which affects, on the one hand, the color tone for the sections and, on the other hand, the shown metrics in the details dialog) was always the test set that covers the root of the sunburst diagram (i.e., the whole project). Especially when navigating through the diagram in mode *Focusable*, this resulted in very small coverage density values when focusing, e.g., on a package on lower layers (which resulted in similar issues as described earlier concerning the coloring factor). It also felt unnatural that the reference test set contains tests that do not even cover the currently focused section. Therefore, it came out to be wiser to always select the test set covering the currently innermost section of the diagram as the current reference. This leads to the effect, that the metric calculations are always different depending on the currently focused package and specifically only the tests covering the latter are taken into account.

6.3.2 Code View

Figure 6.7 shows the implemented code view for a class *at.simianarmy.service.MembershipService* within an example project, which satisfies requirement *R4*. The view is divided into two areas. The left side shows the actual code of the current class together with a number indicating how many test cases cover the respective line. Each line is highlighted with a color tone corresponding to the coverage density on LOC level, whereby the reference test set is always the sum of tests covering the shown class. The right side is reserved for the detail popup dialog, which contains a list of the covering test cases and appears when the user clicks on a specific LOC. The info buttons lead to the respective test views.

Line Coverage Overview Class at.simianarmy.service.MembershipService

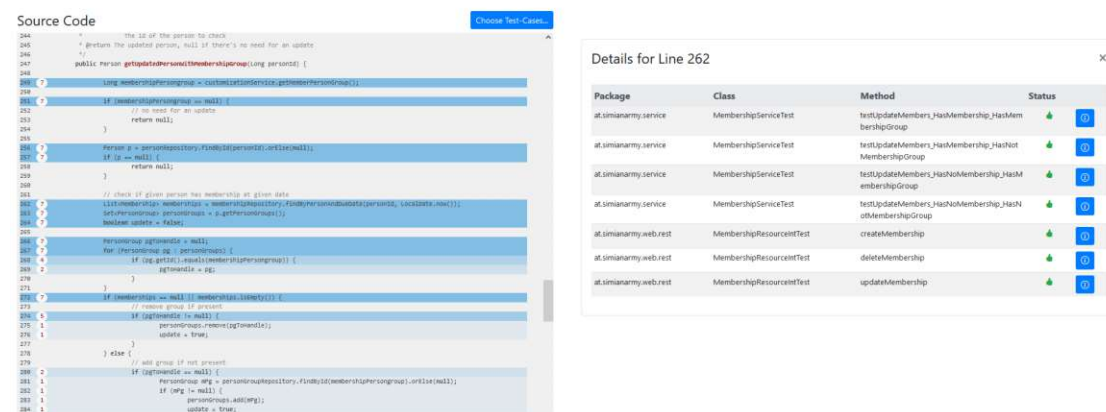


Figure 6.7: Coverage Density Report: Code View

By clicking on the button *Choose Test-Cases...* the user may adapt the reference test set, which is taken into account (requirement *R5*). This is done via a modal dialog, which offers the possibility to add or remove test classes and test methods (Figure 6.8). Note that the dialog got enhanced with additional filter possibilities during the implementation phase. In detail, search bars for test classes and test methods were added to enable filtering and easier navigation through the possible test cases.

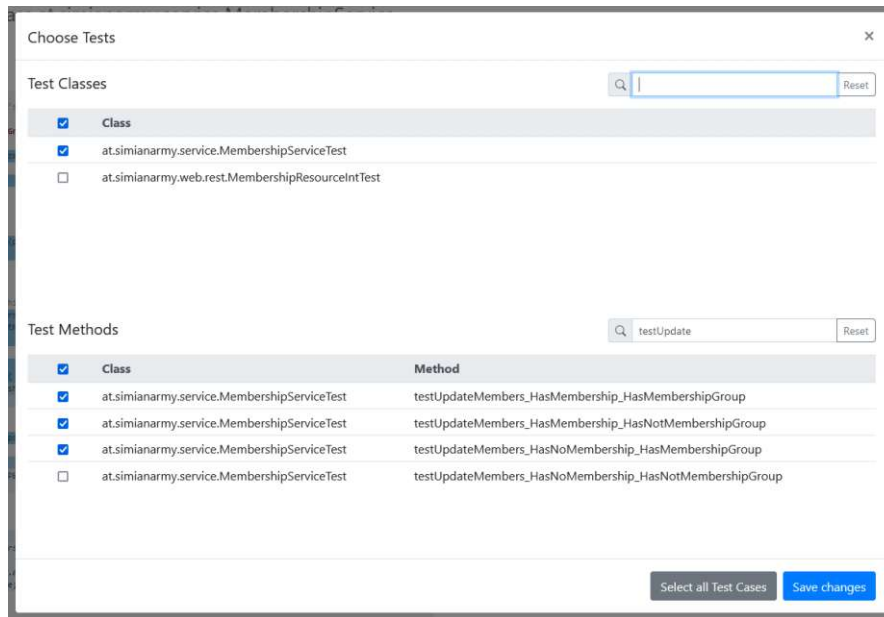


Figure 6.8: Coverage Density Report: Test Selection

6.3.3 Test View

Figure 6.9 shows the test view for a test class *at.simianarmy.web.rest.MembershipResourceIntTest* within an example project (requirement *R7*). The view presents the classes and methods covered by the respective test class, as well as its source code. Furthermore, the overall status of the test class is visualized through green (passed) or red (failed) thumbs up signs. In addition, the number of passed (referred to as *Successful Tests* in this visualization) and failed tests is displayed. In the source code view, the execution status for each test method is also displayed individually. The info buttons in the list of covered classes lead to the respective class view.

Test Class Details at.simianarmy.web.rest.MembershipResourceIntTest

Status: ✔
 Successful Tests: 7
 Failed Tests: 0

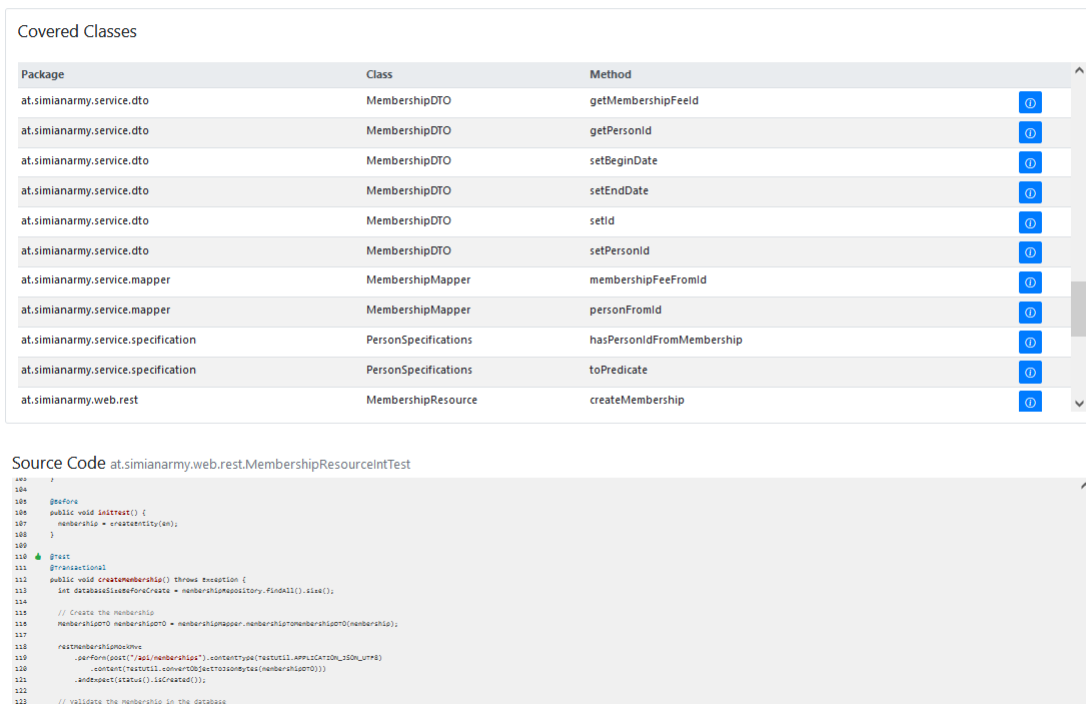


Figure 6.9: Coverage Density Report: Test View

6.3.4 Reusability and Integration

Concerning the non-functional requirements $N1$ and $N2$, which are defined in Section 5.3.3, it needs to be held that they are fulfilled implicitly by the chosen architecture of the prototype.

Concerning $N1$, the files `data/data.js` and `js/coverage-density-functions.js` can be re-used independently from the rest of the report's files as they contain all the necessary information and computation steps for the coverage density metrics on all levels (see also Section 6.2.3). Therefore, it is possible to extract those two files and use them as a basis for visualizing the metric in other tools and reports.

Furthermore, as the report is an independent and static HTML website, it can easily be re-used by build servers that offer HTML report embedding. For example, *Jenkins* and *Atlassian Bamboo* provide this functionality by publishing generated HTML reports in their build job results [58], [49].



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Evaluation

This chapter describes the conducted evaluation of the ideas and concepts established during this thesis. To achieve this, a scenario-based expert evaluation was conducted with the aid of a coverage density report for an example project generated by the prototype described in Section 6. Section 7.1 states fundamental aspects that were considered before the actual evaluation was planned and the scenarios were designed. In Section 7.2, the example project which formed the basis for the generated coverage density report used during the evaluation will be presented. The Sections 7.3 and 7.4 describe the overall procedure and the demographic information about the participants. The designed scenarios are presented in Section 7.5, whereas the results of the conducted evaluation will be depicted in Section 7.6. Finally, those results will be interpreted in regard to the research questions in Section 7.7.

7.1 Fundamental Considerations

When evaluating concepts or metrics through an implemented visualization prototype, the participants should conduct predefined tasks using the latter, get observed by a supervisor in order to record how it aids in fulfilling those and get interviewed about their experiences. For the actual evaluation in this thesis, a coverage density report generated by the prototype as described in Chapter 6 was used for achieving this. As such a report is always generated for exactly one specific software project, it was necessary to investigate on how to choose this base project in order to ensure a consistent and expedient form of evaluation.

In fact, there were two possibilities concerning the election of base projects for the report generation. The participants either could have chosen the project to analyze themselves or been given a pre-generated report for a fixed project. The main aspect to keep in mind when making the decision for one of the above possibilities was that the participants could not conduct realistic tasks without a certain level of knowledge of the base project

for which the report was generated. A simplified example for this would be a task where a participant needs to state if a class is covered by unit tests in an effective and appropriate way. If the participants do not have any knowledge about the actual requirements and functionalities of an implementation, they may be incapable of stating whether the tests covering the code are sufficient.

The natural intention would be to let participants choose a project which they are currently working on and have enough expert knowledge about. Hence, this heavily impedes the definition of the scenarios and tasks to conduct as it implies that each of them must be as general as possible, s.t. it can be fulfilled with every possible software project. As the defined research questions were defined around very specific tasks (such as assessing the risk of a refactoring), this was not feasible in the context of this thesis. Furthermore, the results of the evaluation would not be comparable as they are based on different projects that the report was generated for.

Generating the coverage density report for a fixed project would, on the one hand, ease the construction of scenarios and ensure the comparability of the results but, on the other hand, contradict the previously stated fact that participants need project-related knowledge to fulfill the tasks within the defined scenarios. Nevertheless, the decision was to choose a fixed project as a basis and give the participants the necessary insider knowledge in order to understand the scenarios. The respective project will be briefly described in the following section.

7.2 Example Project

The software project used as a basis for the coverage density report was the open-source software *Walter*¹, which is a management platform for music societies and has been developed by a group of students at the University of Technology in Vienna in the course of several classes. The reason why exactly this project was chosen is that the author of this thesis was one of those students and, therefore, also highly involved in the whole development process. Therefore, designing reasonable scenarios for the evaluation was easily possible with the available expert knowledge the author had. The software was developed for a real customer and is of reasonable size, which makes it a realistic and comprehensive base project for the evaluation (approx. 55k LOC and 950 test cases in the backend; overall effort of approx. 9 person months). Furthermore, the used technology stack corresponds to the current state of the art, which also makes it comparable to other small to medium sized Java projects.

¹<https://gitlab.com/fonkwill/walter>

The main features of *Walter* are as follows:

- person and membership management
- inventory management
- cashbook management
- appointment management
- serial letter generation
- document management (with cloud platform integration)

The software is based on the following technologies:

- *Java 8* and *Spring Boot* (backend)
- *Maven* as the primary build tool for the backend
- *AngularJS* and *Twitter Bootstrap* (frontend)
- *PostgreSQL* (database)

The architecture is a modern web architecture, meaning that the backend acts as a pure REST server, which provides business logic for the business processes and states the main interface for the browser application. The latter is realized as a Single Page Application (SPA) which consumes/sends data from/to the REST endpoints.

Note that for the evaluation only the backend was taken into account. In particular, the coverage density report was generated for the REST server along with its corresponding test suite.

7.3 Procedure

The evaluation sessions were conducted in six separate online video sessions (one session per participant). As it was necessary to investigate on remarks and trains of thought of the participants while acting out the scenarios, the sessions were both audio- and screen-recorded. In addition, the whole process was supported by an online questionnaire created with an online tool beforehand (see Section A.2).

At the beginning of each session, the metric *code coverage density* on the different metric levels (LOC, method, class, package and higher levels) was explained by reasonable examples (Figure A.16). Afterwards, an overview of the main features and the usage of the generated HTML report was given. The focus of the latter was, on the one hand, on drawing lines between the defined metric and the coverage density report (i.e.,

showing how the metric gets visualized in different views) and, on the other hand, on instructing the participants on how to navigate through the report and use it in general. Subsequently, the scenarios were acted out, whereby all the questions defined in Section 7.5 were handled with the online questionnaire (Figures A.17, A.18, A.19 and A.20). In the end, there was a final discussion about general remarks and further special realizations that the participants gained from the visualizations of the report.

Note that in each session, the conductor shared his screen to the participant and allowed remote control on his own computer. The report, as well as the online questionnaire, were therefore both running and accessed on the conductor's device and did not have to be forwarded to the participants. This approach supported the process in an optimal way as the actions within the report had to be taken by the conductor (i.e., the introduction of the prototypes functionalities) as well as the participants (i.e., the usage of the report during the scenarios).

The overall procedure, the presented scenarios and the questionnaire were furthermore evaluated through a preceding pilot session. The intention was to check, if on the one hand, the scenarios are reasonable and well-defined and, on the other hand, if the given introductions concerning the metric definition and the report usage were adequate for acting them out. Nevertheless, the target was also to verify the accompanying questions, i.e., to check whether they were misleading or lacking important information in order to get a consistent set of answers. Based on this pilot session, the overall procedure, the scenarios and questions were adopted respectively.

7.4 Participants

The participants were six experts from the domain of software engineering. Four of them were between 25 and 34 and the other two between 35 and 50 years old. All interviewees had experience in the industrial area and half of them was also working in the scientific area. The respective research fields of those three participants were *Software Engineering*, *Machine Learning* and *Agile Distributed Software Development*. Five participants had five to ten years of experience in the industrial area, whereas one participant had eleven to 20 years of experience. Concerning the participants with experience in the scientific area, one had less than five years and two between five and ten years of experience.

7.5 Scenarios

In the following, the designed scenarios are explained in detail. Within each scenario, the participants were asked a set of quantitative and qualitative questions. As the outcomes of the evaluation of those questions should in the end support the answering of the defined research questions, the scenario questions were additionally categorized by their corresponding research question.

7.5.1 Frame Story

In order to achieve a suitable factor of realism during the conduction of the evaluation, the scenarios were designed around an accompanying frame story. The participants should imagine that they are software engineers and supporters of the music society *Stadtkapelle Hainburg/Musikverein Wolfsthal* and contribute on their management application *Walter*. They were told that during the last team meeting with their colleagues, it was discussed that the software quality in the backend should be analyzed and increased concerning maintainability, code quality and test suite quality. For supporting this intention, they were provided a prototype of a novel coverage analysis tool which generates a static HTML report and visualizes the metric *Code Coverage Density* on different abstraction layers. For the current version of the backend, the respective report has already been generated with this tool and should be used for further analysis (i.e., for acting out the scenarios).

7.5.2 Scenario 1

In the first scenario, the participants should get a first overview of the general coverage of the software, as well as the distribution of the test cases. For this, they should use the sunburst diagram with the setting *Show all Levels*. It was also pointed out that they may change the configurations (i.e., scaling and coloring factor) individually such that they get an optimal overview of the project and the coverage. The questions asked were as follows (the corresponding research questions are stated in square brackets):

Q1: Are there any packages or classes, that stand out most, e.g., because of an unbalanced test coverage or their coverage density values? [RQ2]

Q2: How evenly are the test cases distributed within the package *at.simianarmy.service* in your opinion (on a 5-point scale, where 1 = *not evenly distributed* and 5 = *evenly distributed*)? [RQ2]

Q3: How well do the data and the visualizations support you for getting a first overview over the test case distribution (on a 5-point scale, where 1 = *very poor support* and 5 = *very good support*)? [RQ2]

7.5.3 Scenario 2

In the second scenario, the target was to weight up the risk of a refactoring of a specific class within the project (*at.simianarmy.letter.SerialLetterWorker*). The participants were first given the necessary information about the requirements that the implementation of the class covers. They were told that the code is highly complex and not maintainable in the current state. Therefore, it should undergo a refactoring in order to increase the maintainability, whereby concrete refactoring steps have not been defined yet. It was clarified that the focus is not on stating how to refactor the class but on assessing the potential risk of a refactoring. Furthermore, the conductor presented and re-explained the visualizations given on package level (i.e., the sunburst diagram for the package

at.simianarmy.letter) and on code level (i.e., the code view for the respective class). The questions asked were as follows:

Q4: What is your estimate concerning the code coverage of this class (on a 5-point scale, where 1 = *very bad coverage* and 5 = *very good coverage*)? [RQ2]

Q5: How evenly are the test cases distributed across the class in your opinion (on a 5-point scale, where 1 = *not evenly distributed* and 5 = *evenly distributed*)? [RQ2]

Q6: How well do the data and the visualizations support you concerning the assessment of the risk of a refactoring (on a 5-point scale, where 1 = *very poor support* and 5 = *very good support*)? [RQ1]

Q7: Based on the results shown in the report: Is it necessary to write further test cases before a refactoring can be performed in your opinion? If yes, please give concrete proposals where code coverage should be increased (i.e., which lines of code or methods should be assessed by further tests). Please do not only consider lines which are covered or uncovered, but also regard the test case distribution. [RQ2]

Concerning question Q7, the participants were furthermore asked to reflect the coverage of the method *createPageEventHandler* (Figure 7.1). The intention was to let them investigate on the significantly differing coverage density measured within this method.

Line Coverage Overview Class *at.simianarmy.letter.worker.SerialLetterWorker*

```

Source Code
Choose Test-Cases...

292  /**
293   * sets the margins and the template in the custom PDFEventHandler
294   *
295   * @param template
296   *       The concerning template to this letter
297   * @param writer
298   *       The writer for the PDF
299   * @return A PageEventHandler for this letter
300   */
301  protected SerialLetterPageEventHandler createPageEventHandler(template template, PdfWriter writer) {
302  13  float marginBottom = SerialLetterConstants.defaultMarginBottom;
303  13  float marginTop = SerialLetterConstants.defaultMarginTop;
304  13  float marginLeft = SerialLetterConstants.defaultMarginLeft;
305  13  float marginRight = SerialLetterConstants.defaultMarginRight;
306  13  PdfImportedPage page = null;
307
308  // read attributes from the template
309  13  if (template != null) {
310  //
311  4  if (template.getMarginBottom() != null) {
312  1  marginBottom = Utilities.millimetersToPoints(template.getMarginBottom() * 10);
313  }
314  4  if (template.getMarginTop() != null) {
315  1  marginTop = Utilities.millimetersToPoints(template.getMarginTop() * 10);
316  }
317  4  if (template.getMarginLeft() != null) {
318  1  marginLeft = Utilities.millimeterToPoints(template.getMarginLeft() * 10);
319  }
320  4  if (template.getMarginRight() != null) {
321  1  marginRight = Utilities.millimetersToPoints(template.getMarginRight() * 10);
322  }
323  }
324
325  // Get the template as PDF page
326  4  PdfReader reader = null;
327  4  try {
328  4  if (letter.getTemplate().getFile() != null && templatePageResource != null) {
329  3  reader = new PdfReader(templatePageResource.getInputStream());
330  3  page = writer.getImportedPage(reader, 1);
331  }
332  } catch (Exception e) {
333  logger.error("Could not read the template for serialLetter {}", letter, e);
334  throw new SerialLetterGenerationException(ServiceErrorConstants.serialLetterTemplateNotReadable, null);
335  }
336  }
337  }
338  }

```

Figure 7.1: Method “createPageEventHandler” in class view for “SerialLetterWorker”

7.5.4 Scenario 3

In scenario 3, the target was to give an assessment concerning the coverage of a specific package *at.simianarmy.service.specification* (Figure 7.2) through unit, integration and system tests. As a first step, the participants got a brief explanation about what the implemented classes within this package are intended for.



Figure 7.2: Sunburst view for package “at.simianarmy.specification”

In fact, the developers of the project *Walter* implemented complex data queries within the system with Springs *Data Repositories*² and *Specifications*³. The target of the latter is to capsule query-specific code into specification classes which are therefore separated from calls to Springs data repositories. The specification classes follow the builder pattern and therefore, act like query builders. The clear advantage of this feature is that such classes can be tested easier than plain queries (e.g., JPQL) within repository classes, as they follow an object-oriented implementation style. The conductor also introduced this feature to the participants by showing and explaining one specific specification class within the example project (Figure 7.3).

²<https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html>

³<https://spring.io/blog/2011/04/26/advanced-spring-data-jpa-specifications-and-querydsl/>

```

56     * add title checks to specifications.
57     *
58     * @param title
59     *     the title
60     * @return itself for chaining purposes
61     */
62     public CashbookEntrySpecification hasTitle(String title) {
63         4 if (title != null) {
64             2 Specification<CashbookEntry> specification = new Specification<CashbookEntry>() {
65                 /**
66                  *
67                  */
68                 private static final long serialVersionUID = -5397741166431183751L;
69
70                 @Override
71                 public Predicate toPredicate(Root<CashbookEntry> root, CriteriaQuery<?> query,
72                     CriteriaBuilder builder) {
73                     2 return builder.like(builder.lower(root.get("title")), "%" + title.toLowerCase() + "%");
74                 }
75             };
76
77             2 this.specifications = this.specifications.and(specification);
78         }
79         4 return this;
80     }
81
82     /**
83     * add category checks to specifications.
84     *
85     * @param categoryId
86     *     the id of the category
87     * @return itself for chaining purposes
88     */
89     public CashbookEntrySpecification hasCategory(Long categoryId) {
90         2 if (categoryId != null) {
91             1 Specification<CashbookEntry> specification = new Specification<CashbookEntry>() {
92                 /**
93                  *
94                  */
95                 private static final long serialVersionUID = -4769221657555429285L;
96
97                 @Override
98                 public Predicate toPredicate(Root<CashbookEntry> root, CriteriaQuery<?> query,
99                     CriteriaBuilder builder) {
100                    1 return builder.equal(root.get("cashbookCategory").get("id"), categoryId);
101                }
102            };
103
104            1 this.specifications = this.specifications.and(specification);
105        }

```

Figure 7.3: Example specification class (code snippet)

The participants were furthermore told that the package under review in this scenario contains exclusively such specification classes. In addition, the conductor clarified that one main target is that as much coverage as possible for those classes should be received through unit tests and that coverage through integration and system tests is secondary. This means in more detail that a specification class should be considered as covered in an appropriate way if it is effectively covered by unit tests. In order to be able to distinguish between unit, integration and system tests, the participants got the information that unit tests always reside in the same package as the unit under test (i.e., within the package *at.simianarmy.service.specification*), whereas integration or system tests may reside in other packages (e.g., in packages that contain service classes). First of all, the participants were asked the following two questions:

Q8: What is your estimate concerning the code coverage of this package (on a 5-point scale, where 1 = *very bad coverage* and 5 = *very good coverage*)? Please do not only consider the coverage on statement level, but also regard different test levels (unit, integration and system tests). [RQ2]

Q9: How evenly are the test cases distributed across the package in your opinion (on a 5-point scale, where 1 = *not evenly distributed* and 5 = *evenly distributed*)? [RQ2]

As a second step, specific analysis steps were proposed by the conductor. The participants attention was first drawn to the detail views for the following classes:

- *CompositionSpecification* (Figure 7.4a)
- *ClothingSpecification* (Figure 7.4b)
- *InstrumentSpecification* (Figure 7.4c)

Afterwards, the conductor pointed out that *CompositionSpecification* was covered through unit test cases within the package *at.simianarmy.service.specification* and the test class *CompositionSpecificationTest*. The other two classes are not covered through concrete unit tests. Hence, they are covered through integration tests (i.e., tests implemented in the package *at.simianarmy.service* and *at.simianarmy.repository*) or indirectly through tests of the REST endpoint (i.e., through tests implemented in the package *at.simianarmy.web.rest.InstrumentResourceIntTest*).

Based on this analysis, the conductor proposed two conclusions to the participants and asked them whether they agreed with the statements made.

Conclusion A was that the classes *ClothingSpecification* and *InstrumentSpecification* do not have a sufficient test coverage, as the implementation just “gets touched” indirectly through integration tests (in contrast to *CompositionSpecification*).

Conclusion B was that the classes *ClothingSpecification* and *InstrumentSpecification* should be covered by separate and independent unit tests.

The question asked per conclusion was as follows:

Q10/Q11: How far do you agree with this conclusion (on a 5-point scale, where 1 = *no agreement* and 5 = *high agreement*)? [RQ2]

In the end of scenario 3, the participants were asked the following two concluding questions:

Q12: How well do the data and the visualizations support you concerning the discovery of insufficiently covered code parts (i.e., concerning unit, integration and system tests; on a 5-point scale, where 1 = *very poor support* and 5 = *very good support*)? [RQ2]

Q13: How well do the data and the visualizations support you concerning the question which types of test cases (unit, integration or system tests) should be added (on a 5-point scale, where 1 = *very poor support* and 5 = *very good support*)? [RQ2]

7. EVALUATION

Details for Class "at.simianarmy.service.specification.CompositionSpecification" 🔍 ×

<p>Overall Statistics</p> <p>Overall Coverage: 100.00% Method Coverage: 100.00% Statement Coverage: 100.00% Branch Coverage: 100.00% Number of Tests covering this Class: 29</p>	<p>Coverage Density</p> <p>Mean Coverage Density: 4.62 of 104 Tests* Minimum Coverage Density: 0 of 104 Tests* Maximum Coverage Density: 29 of 104 Tests* * Reference Test Set = all Tests covering "at.simianarmy.service.specification"</p>
---	---

Covering Tests

Package	Class	Method	Status
at.simianarmy.service.specification	CompositionSpecificationTest	testFindAll	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testFindComplex	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testFindComplex2	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testFindComplex3	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testFindComplex4	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testFindComplex5	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testFindNothing	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testHasArranger	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testHasArrangerNull	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testHasColorCode	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testHasColorCodeNull	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testHasComposer	🟢 🔍
at.simianarmy.service.specification	CompositionSpecificationTest	testHasComposerNull	🟢 🔍

(a) Details for class "CompositionSpecification"

Details for Class "at.simianarmy.service.specification.ClothingSpecification" 🔍 ×

<p>Overall Statistics</p> <p>Overall Coverage: 76.10% Method Coverage: 75.00% Statement Coverage: 78.30% Branch Coverage: 70.00% Number of Tests covering this Class: 8</p>	<p>Coverage Density</p> <p>Mean Coverage Density: 3.47 of 104 Tests* Minimum Coverage Density: 0 of 104 Tests* Maximum Coverage Density: 7 of 104 Tests* * Reference Test Set = all Tests covering "at.simianarmy.service.specification"</p>
--	--

Covering Tests

Package	Class	Method	Status
at.simianarmy.repository	ClothingRepositoryTest	getAirWithNoPerson	🟢 🔍
at.simianarmy.repository	ClothingRepositoryTest	getFreeClothingsWithBeginDateAndEndDateOfAnyTypeAndSize	🟢 🔍
at.simianarmy.repository	ClothingRepositoryTest	getFreeClothingsWithBeginDateBeforeAllSavedOfAnyTypeAndSize	🟢 🔍
at.simianarmy.repository	ClothingRepositoryTest	getFreeClothingsWithBeginDateOfAnyTypeAndSize	🟢 🔍
at.simianarmy.service	ClothingGroupServiceTest	testDelete	🟢 🔍
at.simianarmy.service	ClothingGroupServiceTest	testDeleteMoreThanAvailable	🟢 🔍
at.simianarmy.service	ClothingGroupServiceTest	testGetAll	🟢 🔍
at.simianarmy.service	ClothingGroupServiceTest	testSave	🟢 🔍

(b) Details for class "ClothingSpecification"

Details for Class "at.simianarmy.service.specification.InstrumentSpecification" 🔍 ×

<p>Overall Statistics</p> <p>Overall Coverage: 41.40% Method Coverage: 64.30% Statement Coverage: 39.50% Branch Coverage: 27.80% Number of Tests covering this Class: 1</p>	<p>Coverage Density</p> <p>Mean Coverage Density: 0.39 of 104 Tests* Minimum Coverage Density: 0 of 104 Tests* Maximum Coverage Density: 1 of 104 Tests* * Reference Test Set = all Tests covering "at.simianarmy.service.specification"</p>
--	--

Covering Tests

Package	Class	Method	Status
at.simianarmy.web.rest	InstrumentResourceIntTest	getAllInstruments	🟢 🔍

(c) Details for class "InstrumentSpecification"

Figure 7.4: Details for various specification classes

7.5.5 Scenario 4

The last scenario focused on finding and eliminating test case redundancies with the provided report. For this purpose, the class *VerificationLinkQualifier* within the package *at.simianarmy.service.mapper.qualifier* was considered. It contains a single method *verificationLinkForPersonId* which generates a verification link for a given user identification with respect to further system configuration (e.g., the servers base URL or whether SSL should be used or not). The reports class view for this class is shown in Figure 7.5.

```

9
10 @Component
11 public class VerificationLinkQualifier {
12
13     @Inject
14     private ServerProperties serverProperties;
15
16     @Inject
17     private WalterProperties walterProperties;
18
19     @Inject
20     private TenantIdentifierResolver tenantIdentifierResolver;
21
22     @Named("verificationLinkForPersonId")
23     public String verificationLinkForPersonId(Long personId) {
24
25         String host = walterProperties.getEmailVerificationHost();
26         Integer port = serverProperties.getPort();
27
28         String protocol = "http";
29
30         if (port == 443) {
31             protocol = "https";
32         } else {
33             protocol = "http";
34         }
35
36         String tenant = null;
37
38         if (walterProperties.getMultitenancy().isEnabled()) {
39             tenant = tenantIdentifierResolver.resolveCurrentTenantIdentifier();
40         }
41
42         String emailVerificationUrl = walterProperties.getEmailVerificationUrl();
43
44         String url = "";
45         url += protocol + "://" + host + "/" + emailVerificationUrl + "?id=" + personId;
46
47         if (tenant != null) {
48             url += "&tenant=" + tenant;
49         }
50
51         return url;
52     }
53
54 }

```

Figure 7.5: Class view for the class “VerificationLinkQualifier”

As in scenario 3, the conductor led the participants through different analysis steps and proposed a conclusion for which the participants should state whether they agree or not. Following to the detail view in the sunburst diagram, the class has a code coverage of 100% and is covered by exactly four test cases (Figure 7.6):

- `testVerificationLinkForPersonId`
- `testVerificationLinkForPersonIdWithSSL`
- `testVerificationLinkForPersonIdWithTenant`
- `testVerificationLinkForPersonIdWithTenantAndSSL`

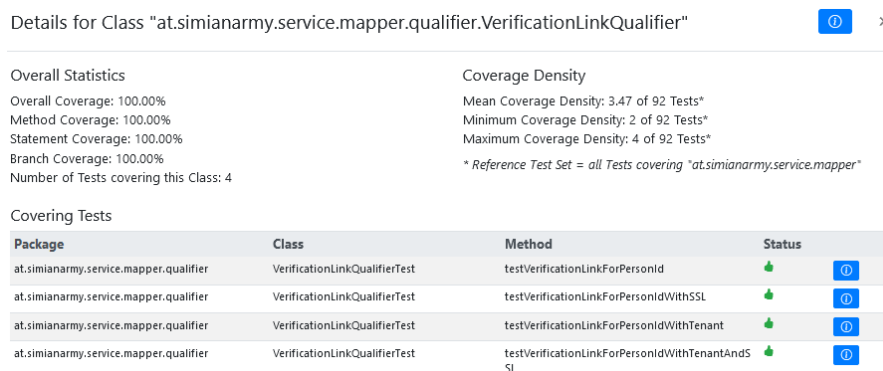


Figure 7.6: Detail view for the class “VerificationLinkQualifier”

The conductor emphasized that the naming of the fourth test leads to the premonition that it might be a duplicate of the test cases `testVerificationLinkForPersonWithSSL` and `testVerificationLinkForPersonWithTenant`. Therefore, the participants attention was drawn to the code view in Figure 7.5 in order to do further investigation on how those tests cover the code exactly. The conductor showed that by adding and hiding test cases within this code view (i.e., by choosing the test cases to be highlighted as described in Section 6.3.2) it gets visible that 100% statement coverage is already reached through the first three test cases. Therefore, the following conclusion was proposed:

Conclusion C: The test case `verificationLinkForPersonWithTenantAndSSL` should be further analyzed concerning test redundancy, as the LOCs of this class are already covered by other tests that cover this class.

Note that when presenting this conclusion, the conductor emphasized in particular that the conclusion does not state that the fourth test case is indeed redundant. It rather expresses that interest on the findings has been awakened and further analysis has to be conducted. The accompanying question was as follows:

Q14: How far do you agree with this conclusion (on a 5-point scale, where 1 = *no agreement* and 5 = *high agreement*)? [RQ3]

Finally, the following concluding question was asked:

Q15: How well do the data and the visualizations support you concerning the search and discovery of test redundancies and duplicates (on a 5-point scale, where 1 = *very poor support* and 5 = *very good support*)? [RQ3]

7.6 Results

In the following, the outcomes of the evaluation are analyzed. The data extracted from the conduction of the evaluation phase can be subdivided into the following groups:

- Group A:** Pure **qualitative data**, which was extracted from the open questions *Q1*, *Q7*, remarks and trains of thoughts that were recorded while acting out the scenarios as well as the final discussion.
- Group B:** Quantitative data concerning the **assessment of facts** by the participants, such as giving an estimate on the code coverage of a package or class (questions *Q2*, *Q4*, *Q5*, *Q8* and *Q9*).
- Group C:** Quantitative data concerning the **agreement on conclusions** proposed by the conductor (questions *Q11*, *Q12* and *Q14*).
- Group D:** Quantitative data concerning the **degree of given support** of the visualizations and the data given by the coverage density report (questions *Q3*, *Q6*, *Q12*, *Q13* and *Q15*, henceforth referred to as *support questions*)

The reason why such a distinction was made is that the data must be interpreted in different ways with respect to the intention of the underlying question it was gathered through.

The qualitative data established for group A was mainly emphasizing and justifying answers and ratings that have been given for questions in all other groups. For the data in group B, the main point of interest was on comparing the estimations given by the participants in order to interpret the similarities of those and imply findings on whether there is a common sense about the results the metric calculation gives and the report visualizes. The same holds for the data in group C. In contrast, for group D it was meaningful to interpret the aggregated data (i.e., the mean ratings) by establishing a ranking of the support level with regards to the different use cases addressed by the asked *support questions*.

The upcoming sections 7.6.1 - 7.6.4 firstly analyze the gathered data in high detail per scenario. A ranking of the support questions is given in Section 7.6.5. The Sections 7.6.6 and 7.6.7 summarize the remaining qualitative data from group A (i.e., improvement suggestions and further remarks).

7.6.1 Scenario 1

Concerning question *Q1*, *at.simianarmy.service*, *at.simianarmy.dto* and *at.simianarmy.rest* were identified most frequently as noticeable packages. It also emerged that there is a common sense that those three packages should be analyzed further. The participants mainly mentioned packages, which stood out because the classes within the packages had highly alternating coloring tones. All of them captured very well that there are significant differences concerning the test case distribution.

Figure 7.7 shows the answers given by the participants for question *Q2*. The given ratings result in a mean rating of 2.33 , whereby it needs to be held that interviewees rather estimated a moderate to bad test case distribution. As there were no participants rating the distribution with a higher value than 3 it can be deduced that they shared this perception.

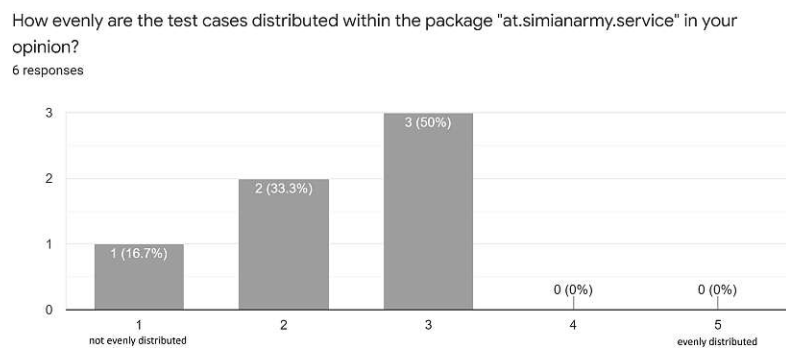


Figure 7.7: Answers for question *Q2* in scenario 1

Concerning the question on how well the data and the visualizations support the participants for getting a first overview over the test case distribution (*Q3*), the calculated mean rating was 4 . Nevertheless, the concrete ratings for this question reached widely from 2 to 5 (Figure 7.8), whereby the lower rankings were justified by the following recorded incidents and trains of thoughts of the participants.

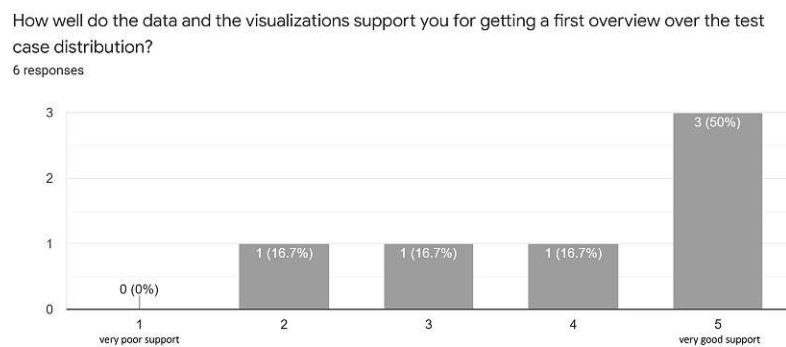


Figure 7.8: Answers for question *Q3* in scenario 1

Some participants had problems with finding an optimal setting for the coloring factor and found sections that seemed equal because of a visually equal color tone but had significantly different coverage density values. Therefore, it appeared that, e.g., sections with a dark color tone should not be considered as equal in the first place. Moreover, one participant explicitly stated that visualizing this metric with one color and different tones may be problematic for two reasons. First, it simply decreases the accessibility for users who suffer from color blindness. Second, it is definitely harder to spot differences in the metric values if color gradations are preferred over using distinct colors.

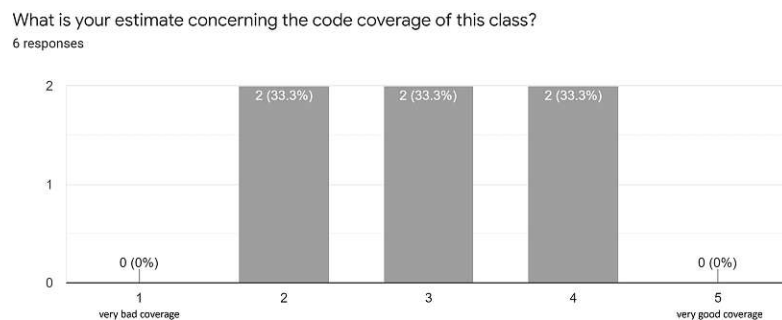
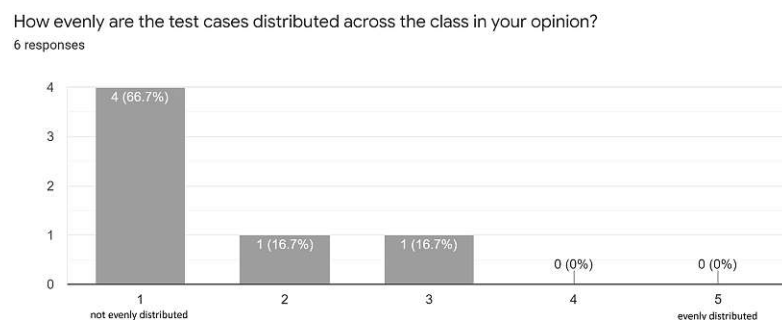
Two participants especially proclaimed some of their thoughts concerning the test set that is taken into account per default when navigating through the sunburst diagram. One of them stated that the coverage density on package level is hard to assess because the value taken into account for the color tone is accumulated and based on the underlying classes, methods and LOCs. In the interviewee's opinion, it is inevitable to additionally investigate the class and code views to get a more precise insight. The other one claimed that considering the coverage density on project level may be suboptimal because there are certainly many small classes, which are naturally covered by less tests than bigger classes. Furthermore, the respective participant stated that navigating through the diagram in mode the *focusable* makes more sense as the reference test set is always the sum of tests that cover the currently focused (innermost) section of the diagram and not the whole test suite.

In addition, some participants also tried to estimate the general code coverage. One interviewee tried to increase the coloring factor as high as possible, searched for sections that were still not colored, and concluded that those may not be covered at all. It came out that spotting uncovered parts of a program may be easier in classical coverage tools. Another participant took this thought even further and claimed that the sunburst diagram visualization is mainly helpful for judging the test case distribution, but not for estimating the general code coverage.

7.6.2 Scenario 2

In scenario 2, the participants showed a good ability to recognize uncovered LOCs with the support of the report's class view. The Figures 7.9 and 7.10 depict the given answers to the questions *Q4* and *Q5*.

The participants rated the code coverage of the respective class with values between 2 and 4, which results in the fact that from the participant's view the coverage is neither *very bad* (1) nor *very good* (5). The mean value is exactly 3 and there were no remarkable outliers. Concerning the test case distribution, most of the participants rated the test cases as *not evenly distributed* (1). Two participants did not share this drastic opinion and rated the distribution with 2 and 3. Nevertheless, the calculated mean value of 1.5 and the fact that there were also no significant outliers show that the estimations are rather towards a non-even test case distribution.

Figure 7.9: Answers for question Q_4 in scenario 2Figure 7.10: Answers for question Q_5 in scenario 2

Concerning question Q_7 , the participants stated that they would write further tests at least for uncovered exception handling and uncovered methods in general. Furthermore, it stood out that there are many LOCs that are covered by 13 test cases. The interviewees remarked that they would conduct further analysis on what those test cases are intended for, as they assumed that this either means that there is a high amount of black box tests or that those 13 test cases are testing the same specifications. The review of the method `createPageEventHandler` also awoke the participants' interests. They wondered especially why there was a sudden decrease of test cases from 13 to four on line 312 (Figure 7.1) and said that they would definitely do further analysis in this area.

Concerning the degree of support that the data and visualization gives, the evaluation of the ratings for question Q_6 showed that the participants found it very helpful for assessing the risk of a refactoring (Figure 7.11, mean value = 4.5). One participant furthermore emphasized that, in the context of a refactoring, it is very useful to know how the test cases are distributed across the code and especially which test cases cover which LOC. Nevertheless, it seems that the concentration is more towards the test case counter than on the color tone for each LOC. In addition, one participant pointed out that giving a highly qualified estimate on the risk of a refactoring for the class under review is hard as it is unknown (not visualized) whether a specific test is a unit, integration or system test.

How well do the data and the visualizations support you concerning the assessment of the risk of a refactoring?

6 responses

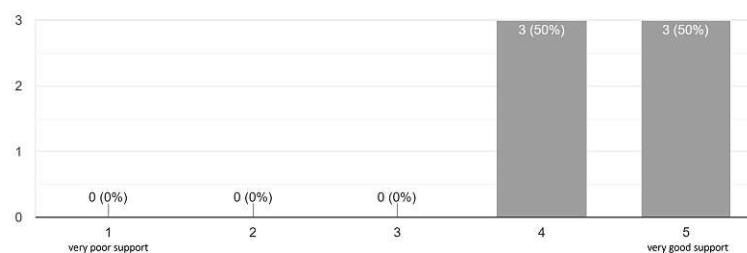


Figure 7.11: Answers for question *Q6* in scenario 2

7.6.3 Scenario 3

Similar to scenario 1, the participants tried to give an estimation about the code coverage and test case distribution of the package *at.simianarmy.service.specification* by relying on the sunburst visualization's color highlighting of the different sections. In order to characterize the test levels (unit, integration and system tests), they reviewed the listed test cases in the detail views for each section (class) within the diagram.

The Figures 7.12 and 7.13 show the given answers for the questions *Q8* and *Q9*. As in the latter scenarios, the participants had similar perceptions concerning the coverage and test case distribution within the analyzed package. The code coverage was rated with a mean value of 2.17 and the distribution with a mean value of 1.67 , whereas there were no remarkable outliers, e.g., ratings with the values 4 or 5 . While the participants were estimating the code coverage and test case distribution, most of them remarked that they are missing a concrete visualization for distinguishing between unit, integration and system tests. Even though the conductor gave the needed domain knowledge in order to fill this gap (i.e., by clarifying that test levels can be characterized by the tests naming convention), the interviewees still had problems to classify the test cases and therefore to give highly valuable estimates for questions *Q8* and *Q9*.

What is your estimate concerning the code coverage of this package? Please do not only consider the coverage on statement level, but also regard different test levels (unit, integration and system tests).

6 responses

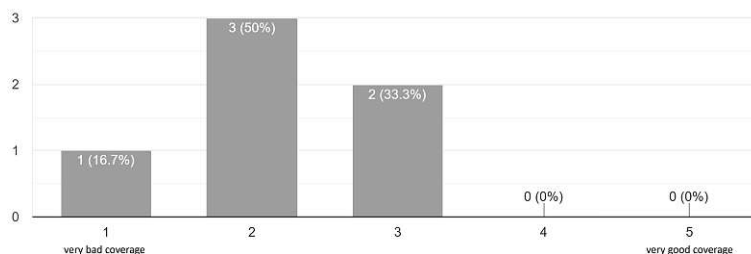


Figure 7.12: Answers for question *Q8* in scenario 3

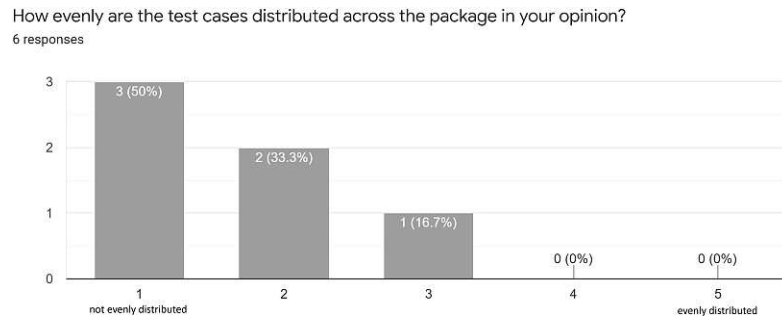


Figure 7.13: Answers for question *Q9* in scenario 3

Concerning the analysis steps proposed by the conductor, the participants mostly agreed with the conclusions stated (Figures 7.14 and 7.15). Nevertheless, for both conclusions A and B, there was one participant who did not consent at all due to the lack of test level visualization. Given the fact that a concrete estimation and analysis was only possible with the domain knowledge deliberately brought in by the conductor, the interviewee felt that those conclusions could not have been drawn on one’s own and therefore disagreed. Concerning conclusion B, there was one participant who rated the degree of agreement with 4, stating that further tests should only be established if it is clear what the code is intended for and additional unit tests make sense.

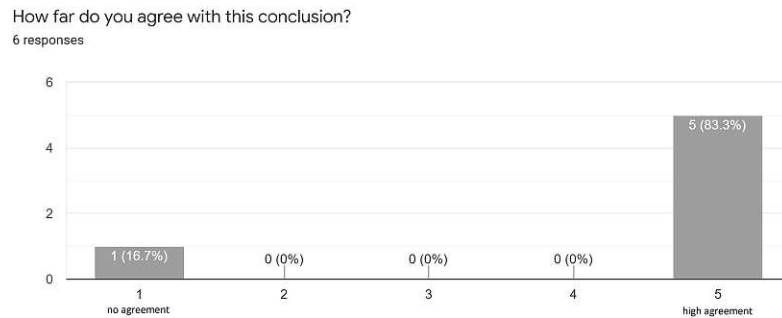


Figure 7.14: Agreements on conclusion A in scenario 3

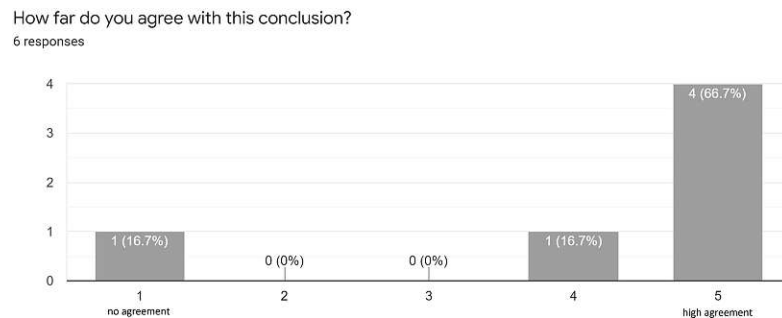


Figure 7.15: Agreements on conclusion B in scenario 3

The Figures 7.16 and 7.17 depict the given ratings for the questions *Q12* and *Q13*, which were focused on the support concerning the discovery of insufficiently tested code parts and the question, which types of tests should be added. The ratings for both topics spread from 1 to 4 (1 = *very poor support*, 5 = *very good support*), whereas all the participants argued their ratings with the circumstance that the test levels are not visualized. The interviewees therefore mostly stated that support is either poor or at least impaired by this missing feature. One participant proclaimed that apart from the missing test level classification, also a filtering mechanism was missing in order to automatically view only tests of a certain type in the sunburst and detail view.

How well do the data and the visualizations support you concerning the discovery of insufficiently covered code parts (i.e. concerning unit, integration and system tests)?

6 responses

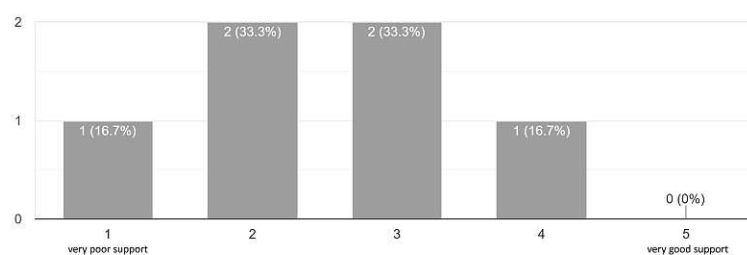


Figure 7.16: Answers for question *Q12* in scenario 3

How well do the data and the visualizations support you concerning the question which types of test cases (unit, integration or system tests) should be added?

6 responses

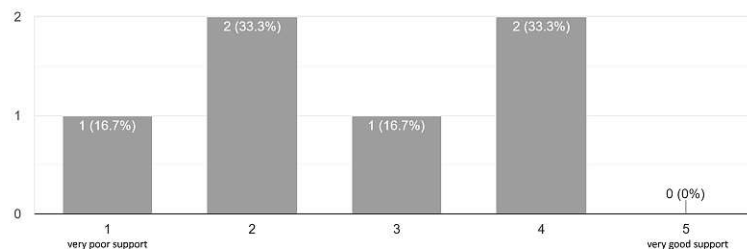


Figure 7.17: Answers for question *Q13* in scenario 3

7.6.4 Scenario 4

Concerning the presented analysis steps and the conclusion provided by the conductor, the participants agreed that the suspected test case could be a redundancy but hence should be further analyzed before it is deleted. Figure 7.18 shows the concrete agreement ratings for conclusion C in this scenario.

Figure 7.19 furthermore depicts the given ratings for the support level concerning the search and discovery of test redundancies and duplicates. Though the participants agreed with the presented conclusion C, the ratings concerning the degree of support in this

context range widely from 2 to 5 (1 = *very poor support* and 5 = *very good support*). The ratings of 2 and 3 were generally justified with the statement that the given showcase was rather obvious. The interviewees stated that the presented report is lacking some feature for searching such redundancies, which could be achieved by, e.g., analyzing the control flows of the test cases. Nevertheless, all of them clarified that if there is a concrete redundancy suspect, then the report's visualization features definitely support further analysis steps (as it is the case in this concrete scenario).

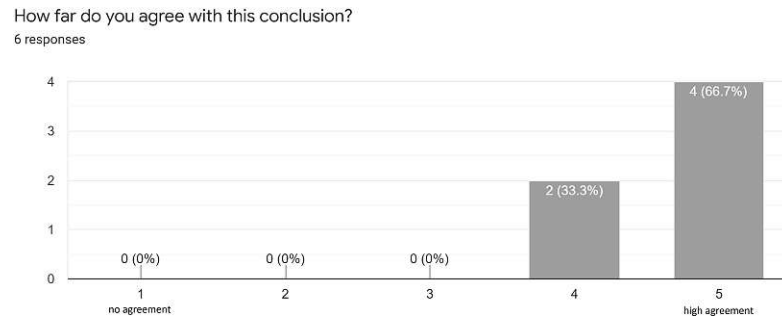


Figure 7.18: Agreements on conclusion C in scenario 4

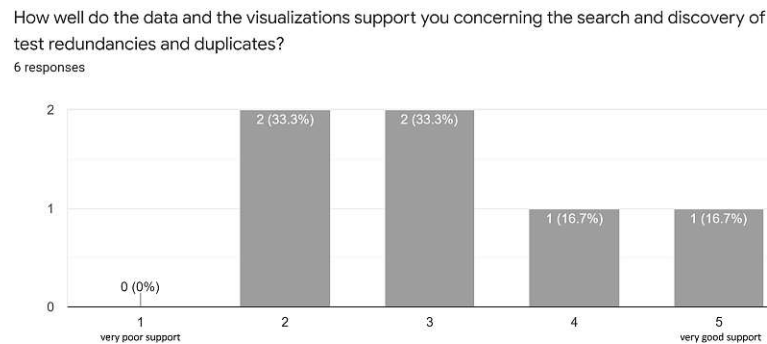


Figure 7.19: Answers for question Q15 in scenario 4

7.6.5 Ranking of Support Questions

Figure 7.20 shows the mean values for each *support question* in a bar chart (1 = *very poor support*, 5 = *very good support*).

The analysis shows that the participants found the report and the data presented most useful for the *assessment of the risk of a refactoring* (mean rating of 4.5), followed by getting a *general overview over the test case distribution* (mean rating of 4.0). The use case *finding test redundancies* has a mean rating of 3.17, which can be explained by the fact that the visualizations and data used in scenario 4 rather give good support in analyzing redundancy suspects and not in searching for redundant test cases (see also Section 7.6.4). Furthermore, the participants seemed to find the visualizations and data less helpful for answering the question *which types of tests should be added* and for

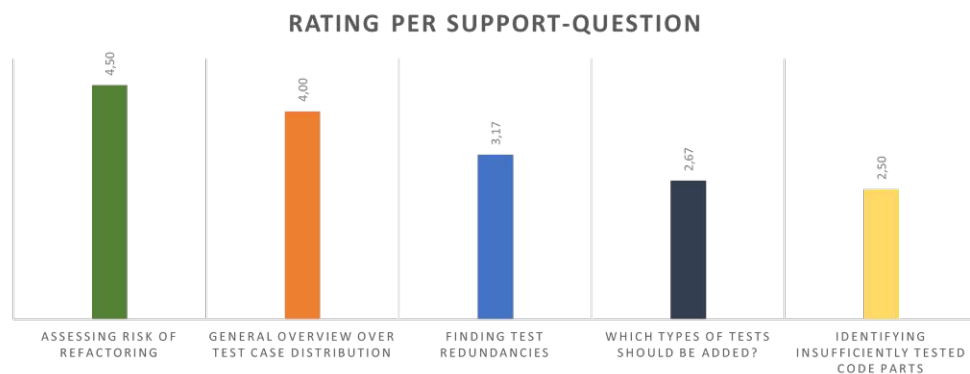


Figure 7.20: Rating per *support question* (1 = *very poor support*, 5 = *very good support*)

identifying insufficiently tested code parts (mean ratings of 2.67 and 2.50). As already described in Section 7.6.3, those lower rankings can be explained by the circumstance that the report does not visualize the test levels yet and therefore distinguishing between unit, integration and system tests is not possible without any domain knowledge in the analyzed software project.

7.6.6 Improvement Suggestions

In this section, the improvement suggestions brought in by the participants for the calculations and visualizations in the report are summarized and discussed.

Sunburst Diagram

Apart from the already discussed qualitative feedback in the previous sections, the participants gave the following inputs for improving the sunburst diagram.

The coverage density of a class in the sunburst diagram could be visualized in more detail considering the underlying coverage density values on LOC level. The idea is to use a color gradient instead of a single-color tone for each class section in order to visualize how the tests distribute over the LOCs within this class. An example of such a possible enhancement is given in Figure 7.21. By following this approach, the user could extract even more information out of the diagram without investigating each section's class diagram.

Establishing categories and subdivisions based on the section's coverage density values (e.g., < 10%, between 10% and 30%, etc.) would furthermore provide additional visualization possibilities in the sunburst diagram. Based on the categorization, the sections could, on the one hand, be labelled by their categories. On the other hand, the sections could be colored with different colors based on their categories instead of a single base color and color tones for the whole diagram.



Figure 7.21: Improvement Suggestion: Color Gradients for Class Sections

Additionally, some participants called for features like filtering and excluding packages and classes. Furthermore, one participant stated that it would be useful if the concrete coverage density value would be visible in a tool tip per section (which gets visible, e.g., on mouse-over). Another participant stated that a legend concerning the color tones would be helpful (e.g., which values correspond to the darkest and lightest color tones).

Class View

For the class view, the participants proposed that stating the sum of tests that cover the currently viewed class would be a good enhancement. Furthermore, one participant mentioned that it would be very helpful if there was the possibility to invert the highlighted LOCs, such that the uncovered lines are marked instead of the covered ones. Another participant remarked that an investigation on the standard deviation of the ClassCovDens for the respective class would be very helpful in order to know whether the calculated mean value is reliable, or the underlying values scatter significantly. The latter statement is very interesting, as in the requirement analysis phase the visualization of this dispersion measure was originally proposed. Hence, during the initial expert interviews it came out that most experts did not estimate this measure as helpful (see Section 5.2). During the evaluation conducted here, it surprisingly emerged that in the context of real world scenarios, there may be a certain degree of additional support that dispersion measures may bring.

Test Level Visualization

As already stated in the latter sections, all the participants called for a visualization that states whether a depicted test case is a unit, integration or system test. Such an automatic classification was suggested for every report view that shows a list of test cases. Nevertheless, the participants also remarked that such an automatic classification is difficult due to several reasons. On the one hand, the developers of the project to analyze may not have followed any guidelines while implementing the tests, which means that

test levels could maybe not be distinguished at all. On the other hand, as for the example project, it is usually the case that unit, integration or system tests are implemented in separate packages or at least follow some predefined naming conventions. Therefore, the participants remarked that at least a classification per configuration through the user should be possible (e.g., by configuring the packages, naming conventions or prefixes that classify the different test levels).

On this basis, several participants also suggested that a mechanism for filtering the test levels within all presented report views would be very useful. Especially in scenario 3, it came out that a high coverage density (and a dark color tone) suggests that there is a high occurrence of tests and that everything seems fine. Nevertheless, such a high density could also be provided through a high coverage through integration and system tests. In the context of this scenario, it would have been helpful to visualize the data only for unit tests in order to immediately recognize where such tests are missing.

Finding Test Redundancies

In order to enhance the prototype with a possible feature for also finding test redundancies (and not only analyzing them), the interviewees suggested that the control flow of the tests could be recorded and taken into account. This basis would then open new possibilities, such as establishing intersection diagrams for execution paths and rankings of test cases based on the similarities of their respective paths. Furthermore, the discussed enhancement concerning test level filtering would also be useful in this context as, e.g., the overlap of tests of a specific test level could be analyzed independently.

7.6.7 General Remarks

In the following, interesting additional general remarks and trains of thoughts that have not already been stated in the latter sections are presented.

Experience with the Metric

Most of the participants emphasized that though they agree that the metric *code coverage density* would in fact establish a useful enhancement of classic code coverage, they simply have not enough experience with it. In detail, this means that many interviewees were not able to distinguish between “good”, “bad” and moreover “desirable” metric values (which already had turned out earlier during the requirement analysis phase and the corresponding expert interviews, see Section 5.2). For other common and long-living metrics, there are, on the one hand, usage guidelines and, on the other hand, so called “good practice values” available that provide support in deciding whether certain quality gates are fulfilled or not. For *code coverage density*, this is simply not the case as the research on this topic is very basic and fundamental. Hence, this established finding builds a good starting point for further research in the future.

Dispersion Measures

As mentioned in Section 7.6.6, one participant especially proclaimed that analyzing dispersion values for the calculated metric would be interesting. In detail, the interviewee stated, that it would be interesting to establish a visualization of the LineCovDens for the LOCs of a class together with the minimum, maximum and mean coverage density values in order to conduct more detailed investigations on the distribution of the test cases. On this basis, it would be possible to recognize, on the one hand, whether the values tend towards the minimum or maximum and, on the other hand, how the values spread around the mean value.

Further Thoughts

In addition to the findings described so far, there were also recognitions that go beyond improvement suggestions, opinions and trains of thought. Some participants also looked at the big picture and thought about what would additionally be possible with the metric and how further research could proceed on this topic.

Referring to the preceding section about the analysis of dispersion measures, the respective participant questioned if the probability of test redundancies is higher when the mean value is close to the maximum value. In other words, the participant thought about whether there is a correlation between the existence of test redundancies and the mean, maximum, as well as the dispersion measures. Another participant stated that from the visualizations it is clearly visible that a high coverage does not mean a high coverage density, which lead to the question whether there is any correlation between those two metrics.

Both ideas should be kept in mind and could form a basis for future work on this topic.

7.7 Discussion

In this section, the results are interpreted in regards to the defined research questions defined in Chapter 1.2. For the sake of traceability, those questions are revised in the following.

RQ 1: How does *code coverage density* support software engineers in deciding how risky a code change or refactoring is?

RQ 2: How does *code coverage density* support software engineers in assessing the code coverage and coverage distribution of a project?

RQ 3: How does *code coverage density* support software engineers in identifying test redundancy and overlaps?

Before answers for those questions are given, some general interpretations of the results are given beforehand.

7.7.1 Similarities of Assessments and Agreements

Summarizing, the answers given by the participants on questions that dealt with the assessment of facts, such as estimating the code coverage or test distribution on given artifacts, were mostly similar in the big picture. Also, the trains of thoughts and justifications, that lead to those ratings, had significant intersections. Those findings lead to the conclusion, that the metric and the presented data, as well as the established visualizations built upon those, lead to highly similar perceptions, considerations and conclusions along all the participants of the evaluation. Moreover, the participants mostly agreed with the analysis steps and conclusions proposed during the scenarios 3 and 4. Hence, there was one participant who absolutely disagreed with the analysis steps and conclusions drawn in scenario 3. Hence, this single outlier can be neglected as those circumstances were clearly traced back to the missing visualization of the test levels, which was nevertheless also remarked by all the other participants, which despite agreed on the conclusions. Therefore, it can be concluded that there was a common sense about the metric *code coverage density* itself, as well as the meaning of the generated data and visualizations.

7.7.2 Support Questions

An aggregation of the support ratings has already been depicted in Section 7.6.5 and Figure 7.20. This bar chart and ranking can be further aggregated in order to match the initial research questions by considering that the questions *Q3*, *Q12* and *Q13* dealt with the same research question *RQ 2*. Therefore, for the final interpretation, the mean value of the mean ratings for those were calculated and a summarizing support rating per research question was established as shown in the bar chart in Figure 7.22.

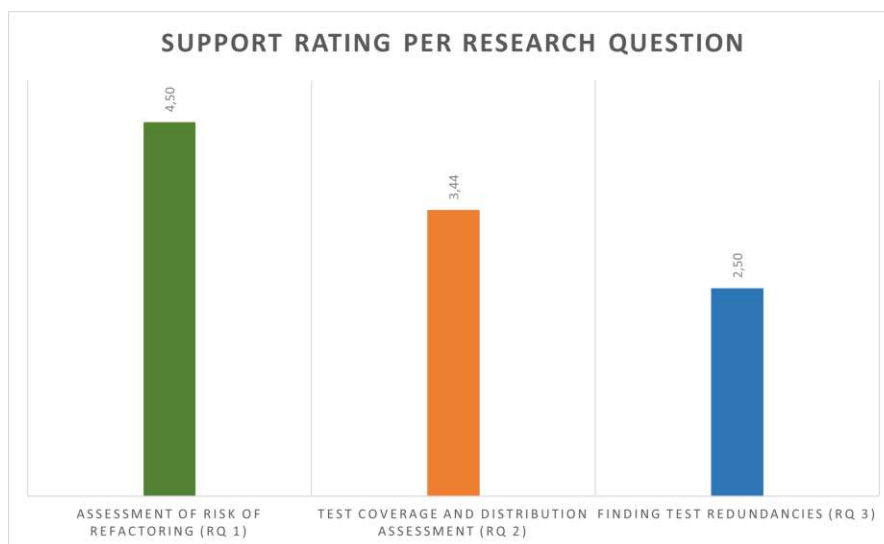


Figure 7.22: Rating per research question (1 = *very poor support*, 5 = *very good support*)

Therefore, the support levels can be ranked as follows:

1. Support for assessment of the risk of a refactoring (mean value = 4.5)
2. Support for assessment of the test coverage and distribution (mean value = 3.44)
3. Support for finding test redundancies (mean value = 2.50)

7.7.3 Answering the Research Questions

Finally, the following conclusions can be drawn concerning the initial research questions.

RQ 1: The metric *code coverage density* supports the process of deciding how risky a code change is very well, as it got the highest mean support level rating of 4.5. Furthermore, the participants stated that the knowledge gained about the test distribution of a class is very helpful in the context of a refactoring. In addition, it emerged that all participants shared similar perceptions about the class presented in scenario 2, which further affirms this interpretation.

RQ 2: The evaluation showed that the metric *code coverage density* has a significant potential for supporting software engineers in assessing code coverage and coverage distribution. The support rating on this topic yielded a mean value of 3.44, which is in the mid-range. Hence, participants argued that the proposed visualizations of the metric need to be enhanced in order to unleash more potential in this area. Furthermore, the metric would need further research concerning “good practice” and “desirable” values.

RQ 3: The evaluation showed that the metric itself rather supports the analysis of already suspected test duplicates, but not finding and identifying those. The support rating on this topic therefore yielded a mean value of 2.50, which is also in the mid-range. Hence, the evaluation yielded an interesting consideration concerning the statistical values computed by the metric. In detail, further research could deal with the question whether there is a correlation between certain constellations of those values and the possibility of test redundancies, which could give further findings on the level of support the metric provides in actually finding and identifying those.

7.7.4 Threats to Validity

Number of Participants

The evaluation of this thesis only used six participants and had both a quantitative and qualitative character. This sample may be too small for generalizing the quantitative results of the evaluation, as this would in general require a larger number of randomly selected participants [50].

Participant Selection

The evaluations participants were mostly members of *Industrial Software (INSO)*, which is a scientific research institution of the University of Technology in Vienna. This could possibly present a selection bias and future research with adopted versions of the established prototype should extend the selection of participants to the industry sector and other organizations.

Example Project

The evaluation was conducted with a coverage density report generated for a specific software project, which therefore could limit the representative character of the results. When considering further evaluation and research with the established prototype, the report should be generated for other and more diverse projects, e.g., for popular open-source projects.

Scenario Design

As described in Section 7.1, it was necessary for certain scenarios to give the participants a reasonable amount of domain knowledge in order to enable them to give suitable assessments. Furthermore, scenario 3 and 4 were constructed in a way such that the conductor led the participants through analysis steps and proposed conclusions, for which the level of agreement was queried. This was necessary, as from the current state of the prototype, it would not have been possible to derive those without the necessary knowledge and proposals. For future work, especially after the proposed enhancements of the prototype, scenarios could be more generalized and designed in a less striking fashion.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Conclusion

This master's thesis showed that though there exists a variety of metrics and tools that support software engineers in the decision-making process during activities in the field of software quality management and assurance, there are still unsatisfied information needs in this context. Therefore, the thesis introduced the novel metric *code coverage density* together with a visualization prototype, that aims at filling those gaps. In the following, the concrete findings and outcomes of this thesis are summarized. Furthermore, an outlook on possible feature work is given.

During the initial scientific research conducted in the first phase, important findings of scientific surveys and investigations in the context of hardly satisfiable information needs have been analyzed and summarized. Furthermore, scientific approaches and tools that try to satisfy them have been discussed. On this basis, currently available coverage tools have been analyzed with respect to the question which of those theoretical findings and concepts have been transferred into tools that are used in practice. The outcome was that from the perspective of the current state of the art (both from a theoretical and practical view), it is still hard to *assess the risk of refactorings*. Furthermore, currently available coverage tools do not put enough emphasis on imparting a *suitable understanding on the relation between source code and test code*. This impedes the support of such tools in the area of *quality assessment of test suites* and *identification of test redundancies*.

Based on those outcomes, it was concluded that further enhancements of classical code coverage analysis and visualizations are needed. For this purpose, the novel metric *code coverage density* which describes how test cases are distributed over specific parts of a software, has been introduced together with feature and visualization proposals that aim at fulfilling the unsatisfied information needs gathered in the first phase. The metric, as well as the feature proposals, were evaluated during expert interviews. The outcomes of those interviews affirmed the relevance of the gathered information needs and confirmed that the metric was well defined and understandable. Furthermore, a set of requirements

for a potential visualization prototype that aims at satisfying the found information needs was established.

Afterwards, the respective prototype which firstly gathers code coverage data from output artifacts generated by *OpenClover* and rearranges it for the purpose of computing the novel metric *code coverage density* and establishing static HTML reports which satisfy the requirements defined earlier, was implemented.

As a last step, the ideas and concepts introduced within this thesis were evaluated. This was achieved by conducting a scenario-based expert evaluation with the aid of a coverage density report generated beforehand with the implemented prototype. The results showed that such an extension of classical code coverage provides significantly better support than existing code coverage tools when it comes to assessing the risk of refactorings. Moreover, the established metric and the visualizations built upon it state a significant potential for imparting a clearer understanding on the relation between source code and test code, which aids in assessing a test suite's quality. Hence, it has been revealed that in the context of test redundancy analysis, those concepts rather support in further analysis of potential redundancy candidates, but not in detecting and locating those.

8.1 Future Work

Last but not least, the evaluation conducted also revealed that the prototype needs some further enhancements in order to unleash more potential. In addition, interesting assumptions and questions aroused which would need further scientific research in the respective context. Potential future work could therefore proceed with the topic of this thesis as follows.

8.1.1 Enhancements of the Prototype

As found during the evaluation, the prototype would definitely need further enhancements with respect to the improvement suggestions summarized in Chapter 7.6.6. A special focus should be kept on a clearer visualization of the test levels (unit, integration or system test) in order to better support the distinction between those. Moreover and though statistical values such as the standard deviation were rated with rather low importance during the expert interviews in the requirements analysis phase, analyzing dispersion values still seems to be interesting.

8.1.2 Reference Values

Moreover, the evaluation showed that concerning the metric *code coverage density*, participants were not able to distinguish between “good”, “bad” and moreover “desirable” metric values. The reason for that is, that there is simply not enough experience with the metric and especially so called “good practice values” are definitely missing as the research conducted in this thesis is very basic and fundamental. Future work could

therefore focus on investigating on desirable reference values for the metric in order to further improve its usability.

8.1.3 Test Redundancy Detection

As found during the evaluation phase of this thesis, the current prototype rather gives support for analyzing test redundancy suspects and is not able to explicitly detect those. Future work could therefore attach to this idea by extending the prototype with features for uncovering duplicated test cases (e.g., by analyzing test execution paths and their similarities).

8.1.4 Integration in other Tools

As the prototype's generated data and established visualizations are easily reusable, potential future work could also engage in incorporating with other tools, such as build servers and continuous integration/build tools in general. In particular, this would enable the generation and evaluation of trend reports in order to gain knowledge about the evolution of the metric values over time and could therefore offer further support in decision-making processes.

8.1.5 Correlations

Finally it needs to be held that some participants stated further thoughts about the correlations of the metric with other facts and circumstances. On the one hand, future work could focus on the question how *code coverage density* relates to classical code coverage and whether there is a correlation between those two metrics. On the other hand, one participant especially thought about the correlation between dispersion measures of the metric (i.e., minimum, maximum, mean and standard deviation) and the probability of test redundancies. In detail, the question that aroused was whether this probability is higher when the mean value is close to the maximum. Future work could therefore draw on this notion and do further scientific investigations in this context. From a more abstract point of view, further research on possible combinations and aggregations with various other metrics (e.g., complexity metrics) could definitely lead to further perceptions on the metric in general (e.g., concerning good practice values).



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Appendix

A.1 Expert Interview Questionnaires

Interview Master Thesis

Thesis Title: Utilizing Code Coverage Density to Enhance Software Quality Management Decisions

<p>How old are you?</p> <p><input type="radio"/> 18 - 24</p> <p><input type="radio"/> 25 - 34</p> <p><input type="radio"/> 35 - 50</p> <p><input type="radio"/> over 50</p>	<p>Are you currently working in the scientific area?</p> <p><input type="radio"/> Yes</p> <p><input type="radio"/> No</p>
<p>Are you currently working in the industrial area?</p> <p><input type="radio"/> Yes</p> <p><input type="radio"/> No</p>	<p>If yes, what is your concrete scientific research field?</p> <hr style="border: 0; border-top: 1px solid #ccc; width: 100%;"/>
<p>If yes, how long have you been working in the industrial area?</p> <p><input type="radio"/> less than 5 Years</p> <p><input type="radio"/> 5 - 10 Years</p> <p><input type="radio"/> 11 - 20 Years</p> <p><input type="radio"/> more than 20 Years</p>	<p>If yes, how long have you been working in the scientific area?</p> <p><input type="radio"/> less than 5 Years</p> <p><input type="radio"/> 5 - 10 Years</p> <p><input type="radio"/> 11 - 20 Years</p> <p><input type="radio"/> more than 20 Years</p>

Figure A.1: Expert Interviews - General Demographic Questions

Metric Definition

Line Coverage Density

The following figure shows the test coverage for a method M1. The method consists of 5 lines of code (Line 1 - Line 5). A subset of tests (T1 - T4) of a test suite is considered. The tests cover the lines as depicted below. "0" means that the respective test does not cover the line and "1" means that it does. The coverage density for a line of code is the number of test cases that cover it. Therefore, the line coverage density "LineCovDens" for line 4 would be 3. In relation to the given subset of tests (which contains 4 tests), it holds that 75% of the test cases cover this line. **How relevant is this information in your opinion?**

Note: you may choose the tests to consider as you wish. The chosen tests may be only a subset of your test suite (which e.g. are of your interest or simply form a logical unit) or the whole test suite.

M1	T1	T2	T3	T4	LineCovDens	CovDens %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%

1
2
3
4
5

not relevant

highly relevant

If you have any remarks on the previous question, please enter them here.

(a) Line Coverage Density

Method Coverage Density

The following figure shows the test coverage of a method "M1". As in the previous example, there are 4 test cases (T1 - T4) which cover the lines of code as depicted in the table. The coverage density for a method is the arithmetic mean value of the coverage density values for the lines that form the method. In our example, the mean value would be 65%, which means that on average 65% of the tests cover this method. **How relevant is this information in your opinion?**

M1	T1	T2	T3	T4	LineCovDens	CovDens %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%
Mean Density M1					2,6	65,00%

1
2
3
4
5

not relevant

highly relevant

If you have any remarks on the previous question, please enter them here.

(b) Method Coverage Density

Figure A.2: Metric evaluation questions (1)

Dispersion Measures

The following figure is an extension of the previous example. In addition to the mean value of the coverage densities of a method's lines of code, we also consider the minimum and maximum coverage densities of the lines. Furthermore, the standard deviation is considered. Those values should clearly state how accurate the mean value (and therefore the coverage density of the method M1) actually is. **How relevant is this information in your opinion?**

M1	T1	T2	T3	T4	Density Count	Density %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%
Mean Density M1					2,6	65,00%
Minim Density M1					1	25,00%
Maximum Density M1					4	100,00%
Std. Deviation Density M1					1,019803903	

1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous question, please enter them here.

(a) Dispersion Measures

Class Coverage Density

The following figure shows the coverage data of two methods M1 and M2, which together form a class "Class A". The coverage density values are already given and computed as explained earlier. The coverage density of a class is (analogous to the computation on method level) the arithmetic mean of the coverage density values of all lines of code, that form that class. In the case below, the coverage density of Class A is the mean of the line coverage densities of Line 1 - Line 12. As for the coverage density on method level, the dispersion measures (minimum, maximum and standard deviation) are also considered here. **How relevant is this information in your opinion?**

Note: the level of abstraction can also be raised to package or module level in analogy to this example.

Class A

M1	T1	T2	T3	T4	LineCovDens	CovDens %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%
Mean Density M1					2,6	65,00%
Minim Density M1					1	25,00%
Maximum Density M1					4	100,00%
Std. Deviation Density M1					1,019803903	
M2	T1	T2	T3	T4	LineCovDens	CovDens %
Line 6	0	1	0	1	2	50,00%
Line 7	1	0	0	1	2	50,00%
Line 8	0	1	1	0	2	50,00%
Line 9	0	1	0	1	2	50,00%
Line 10	0	1	1	0	2	50,00%
Line 11	1	1	1	0	3	75,00%
Line 12	0	1	0	0	1	25,00%
Mean Density M2					2	50,00%
Minim Density M2					1	25,00%
Maximum Density M2					3	75,00%
Std. Deviation Density M2					0,534522484	
Mean Density Class A					2,25	56,25%
Min Density Class A					1	25,00%
Max Density Class A					4	100,00%
Std. Deviation Density Class A					0,829156198	

1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous question, please enter them here.

(b) Class Coverage Density

Figure A.3: Metric evaluation questions (2)

Relevance of Granularity Levels	
How relevant is the notion of coverage density on the level of lines of code in your opinion?	
not relevant	1 2 3 4 5 highly relevant
How relevant is the notion of coverage density on method level in your opinion?	
not relevant	1 2 3 4 5 highly relevant
How relevant is the notion of coverage density on class level in your opinion?	
not relevant	1 2 3 4 5 highly relevant
How relevant is the notion of coverage density on package level in your opinion?	
not relevant	1 2 3 4 5 highly relevant
General Questions	
Is the metric "coverage density" defined clearly and understandable?	<input type="radio"/> Yes <input type="radio"/> No
Do you have any additional remarks concerning the definition and the purpose of this metric?	<hr/>

Figure A.4: Metric evaluation questions (3)

Prototype

Coverage Density Overview (Sunburst Diagram)

The following figure visualizes the metric "coverage density" in a sunburst diagram by showing the distribution of test cases over an application. The diagram is created in accordance to the typical structure of object oriented programs/applications. The inner sections denote modules and packages. The outer sections denote concrete classes. The color gradient visualizes the coverage density of the respective section, where a darker blue tone means a higher emergence of test cases. A brighter tone on the contrary means that there are less covering tests. **How relevant is such a visualization in your opinion?**

not relevant

1
 2
 3
 4
 5

highly relevant

If you have any remarks on the previous question, please enter them here.

Figure A.5: Mockup evaluation questions - Sunburst (1)

For each section in the diagram, it is possible to open a detail view about this section. This view shows overall statistics about coverage and coverage density, as well as which tests cover the selected section. In addition, there is a listing which shows the coverage contribution of the tests concerning the respective section. **How relevant is this information in your opinion?**

Note: all detail views, which show the covering tests for a considered subject (e.g. class) shall mainly support the test suite maintenance and e.g. help to find duplicated tests or test cases which cover code they are not intended to.

Details for Class "Device"

Overall Stats

Overall Coverage: 64.4%	Mean Density: 50% (5 of 10 Tests)
Method Coverage: 55.6%	Minimum Density: 20% (2 of 10 Tests)
Statement Coverage: 60%	Maximum Density: 80% (8 of 10 Tests)
Branch Coverage: 57%	Std. Deviation: 0.5

Tests

Package	Class	Test-Method	Test Status	
test.meat.device.domain	Device Test	testA	OK	show Details
test.meat.device.domain	Device Test	testB	OK	show Details
test.meat.device.domain	Device Test	testC	OK	show Details
test.meat.device.domain	DeviceA Test	testD	Failed	show Details
test.us.menu.controller	UserActionController Test	testActionA	OK	show Details
test.us.menu.controller	UserActionController Test	testActionB	OK	show Details
test.us.menu.controller	UserActionController Test	testActionC	Failed	show Details

Code Coverage Contribution

Package	Class	Test-Method	Contribution
test.meat.device.domain	Device Test	testA	33%
test.meat.device.domain	Device Test	testB	11%
test.meat.device.domain	Device Test	testC	20%
test.meat.device.domain	DeviceA Test	testD	N.A.
test.us.menu.controller	UserActionController Test	testActionA	20%
test.us.menu.controller	UserActionController Test	testActionB	20%
test.us.menu.controller	UserActionController Test	testActionC	N.A.

1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous question, please enter them here.

Figure A.6: Mockup evaluation questions - Sunburst (2)

Class View

The following figure is a visualization of the metric "coverage density" in the context of a Java-Class "rest.device.domain.Device". Each line of code gets an assigned number, which states how many tests cover the respective line. The color gradient visualizes these values. A line with a darker blue tone denotes that there are more tests covering it. A brighter tone denotes a lower emergence of test cases. The visualization's purpose is to highlight parts of the code that are covered through more/less tests. **How relevant is such a visualization in your opinion?**

Line Coverage Overview

Class: rest.device.domain.Device Select Tests

```

1 package rest.device.domain:
2
3 public class Device {
4     public Device() {
5         ...
6     }
7
8     public void execute() {
9         ...
10        ...
11        ...
12        ...
13    }
14
15    public boolean eval() {
16        ...
17        ...
18        ...
19    }
20
21    public boolean destroy() {
22        ...
23        ...
24    }
25
26    public Device copy(Device toCopy) {
27        ...
28        ...
29        ...
30        ...
31    }
32
33 }
```

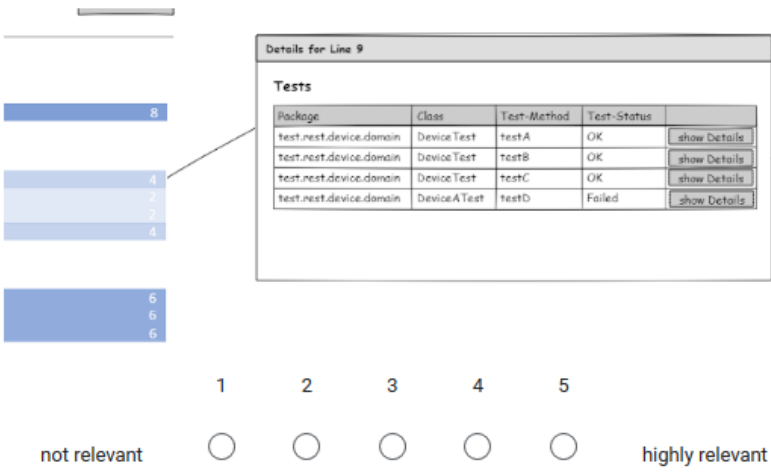
1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous question, please enter them here.

Figure A.7: Mockup evaluation questions - Class View (1)

The following figure is an extension of the previous one. For each line of code, it is possible to view a detailed listing about the covering tests and their state. **How relevant is this functionality in your opinion?**



Package	Class	Test-Method	Test-Status	
test.nest.device.domain	DeviceTest	testA	OK	show Details
test.nest.device.domain	DeviceTest	testB	OK	show Details
test.nest.device.domain	DeviceTest	testC	OK	show Details
test.nest.device.domain	DeviceATest	testD	Failed	show Details

1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous question, please enter them here.

Figure A.8: Mockup evaluation questions - Class View (2)

The test cases that are considered for the visualization may be filtered. Either test methods or even whole classes may filtered in/out. **How relevant is this functionality in your opinion?**

The screenshot shows an IDE window titled 'Class: rest.device.domain.Device' with a 'Select Tests' button. The code in the background is:

```

1 package rest.device.domain;
2
3 public class Device {
4     public Device() {
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

```

The 'Select Tests' dialog box contains two sections:

- Test Classes:** A table with columns 'Class' and a checkbox.

Class	
test.rest.device.domain.DeviceTest	<input checked="" type="checkbox"/>
test.rest.device.domain.DeviceATest	<input checked="" type="checkbox"/>
- Test Methods:** A table with columns 'Class', 'Method', and a checkbox.

Class	Method	
test.rest.device.domain.DeviceTest	testA	<input checked="" type="checkbox"/>
test.rest.device.domain.DeviceTest	testB	<input checked="" type="checkbox"/>
test.rest.device.domain.DeviceTest	testC	<input checked="" type="checkbox"/>
test.rest.device.domain.DeviceATest	testD	<input checked="" type="checkbox"/>

Buttons 'Select All' and 'Save Changes' are located at the bottom right of the dialog.

1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous question, please enter them here.

Figure A.9: Mockup evaluation questions - Class View (3)

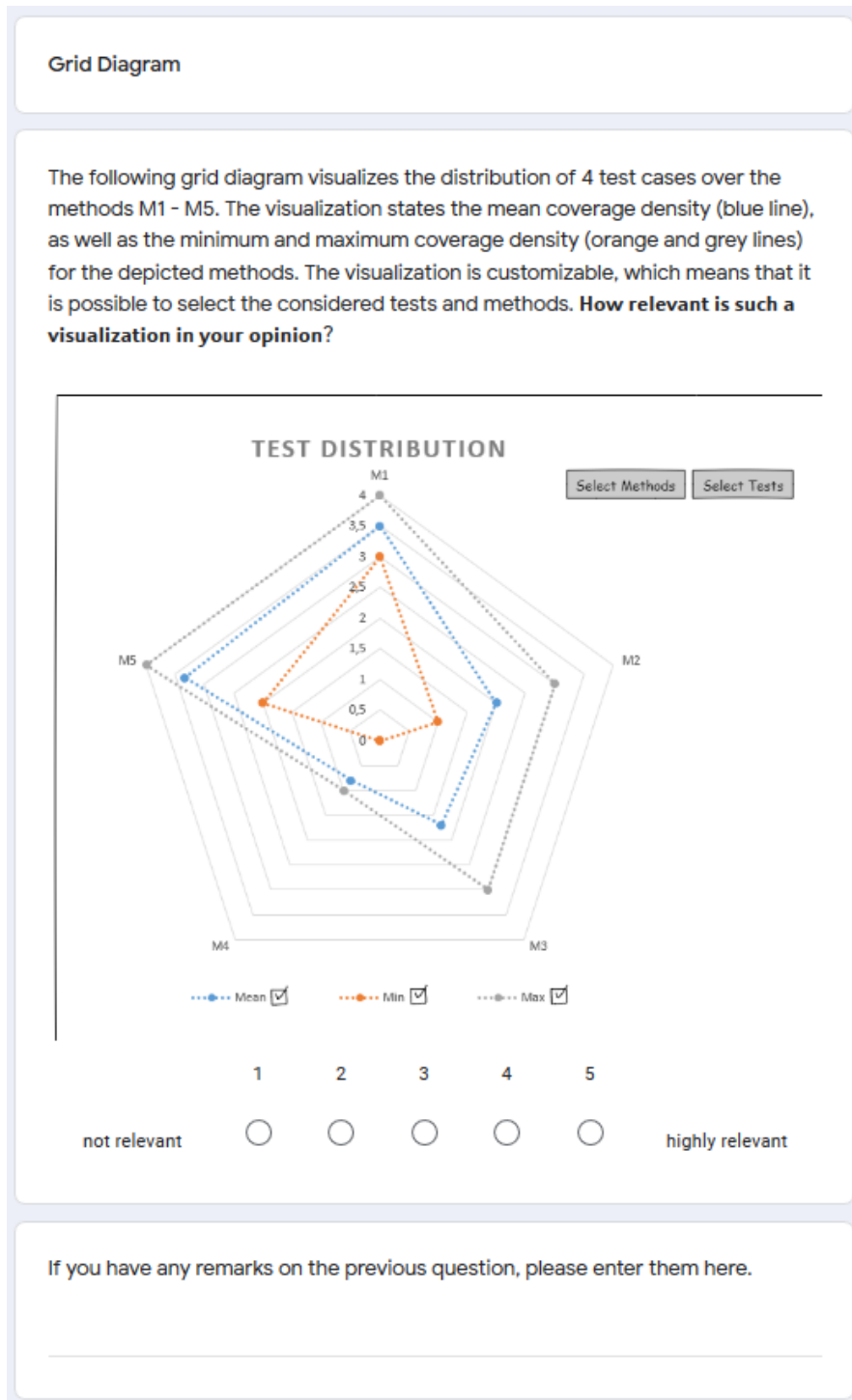


Figure A.10: Mockup evaluation questions - Radar Chart (1)

For each method in the grid diagram, it is possible to open a detail view, which states overall statistics of the method and lists the covering tests. Furthermore, there is a listing of all lines of code that form the respective method, together with their computed coverage density values. **How relevant is this information in your opinion?**

Details for M2 show Method

Overall Stats

Mean Density: 50% (2 of 4 Tests)
 Minimum Density: 25% (1 of 4 Tests)
 Maximum Density: 75% (3 of 4 Tests)
 Std. Deviation: 0.6

Covering Tests

Package	Class	Test-Method	Test-Status	
test	.MethodTest	testA	OK	show Details
test	.MethodTest	testB	OK	show Details
test	.MethodTest	testC	OK	show Details
test	.MethodTest	testD	OK	show Details

1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous question, please enter them here.

Figure A.11: Mockup evaluation questions - Radar Chart (2)

Bar Chart

The following bar chart visualizes the distribution of 4 test cases over the methods M1 - M5. The visualization states the mean coverage density (blue line), as well as the minimum and maximum coverage density (orange and grey lines) for the depicted methods. The visualization is customizable, which means that it is possible to select the considered tests and methods. **How relevant is such a visualization in your opinion?**

Note: a detail view for each method (similar to the view in the previous figure) will also be available in the bar chart.

Test Distribution

Select Methods Select Tests

1

not relevant

2

3

4

5

highly relevant

If you have any remarks on the previous question, please enter them here.

Figure A.12: Mockup evaluation questions - Bar Chart

Test Details

The following figure shows a listing of classes that are covered by a specific test case. In addition, the listing shows the respective coverage contribution of the test. This shall mainly support test suite maintenance, e.g. finding duplicated tests or tests that test code which they are not intended to.

Test Details

Name: test.nest.device.domain.DeviceIntegrationTest
Status: OK

Covered Classes

Package	Class	Method	Coverage Contribution (Class)	
nest.device.domain	Device	execute()	12%	[show Class]
nest.device.domain	Device	eval()	9%	[show Class]
nest.device.domain	Device	destroy	20%	[show Class]
nest.device.domain	DeviceA	destroy	10%	[show Class]
ui.main.menu.controller	UserActionController	commit()	5%	[show Class]
ui.main.menu.controller	UserActionController	evaluateInput()	2%	[show Class]

How relevant is this information in your opinion if we consider that the test is a unit test?

1 2 3 4 5

not relevant highly relevant

How relevant is this information in your opinion if we consider that the test is an integration test?

1 2 3 4 5

not relevant highly relevant

How relevant is this information in your opinion if we consider that the test is a system test?

1 2 3 4 5

not relevant highly relevant

If you have any remarks on the previous questions, please enter them here.

(a) Test Details

(b) Relevance Rating

Figure A.13: Mockup evaluation questions - Test Details

A.2 Evaluation Interview Questionnaires

Evaluation Prototype "Code Coverage Density"

How old are you?

18-24
 25-34
 35-50
 over 50

Are you currently working in the scientific area?

Yes
 No

Are you currently working in the Industrial area?

Yes
 No

If yes, how long have you been working in the industrial area?

less than 5 Years
 5-10 Years
 11-20 Years
 more than 20 Years

If yes, what is your concrete scientific research field?

If yes, how long have you been working in the scientific area?

less than 5 Years
 5-10 Years
 11-20 Years
 more than 20 Years

Figure A.14: Evaluation - General Demographic Questions

Introduction

Initial Situation

You are a software engineer and supporter of the music society "Stadtkapelle Hainburg/Musikverein Wolfsthal" and contributing on their management application "walter". "Walter" is an open source management software for music societies, which has been developed by students together with the customer.

The main features are as follows:

- Person and Membership Management
- Inventory Management
- Cashbook
- Appointment Management
- Serial Letter Generation
- Document Management (with Cloud Platform integration)

The software is based on the following technologies:

- Java + Spring Boot (Backend)
- AngularJS + Twitter Bootstrap (Frontend)
- PostgreSQL (Database)

The backend acts as a REST-Server, which provides Business Logic for the business processes and states the main interface for the frontend application.

During the last project jourfix with your team colleagues, you discussed that the software quality in the backend should be analyzed and increased concerning maintainability, code quality and test suite quality. You are provided a prototype of a novel coverage analysis tool as a support for this intend. This tool, which generates static HTML reports, visualizes the novel metric "Code Coverage Density" on different abstraction layers. For the current version of the backend, the report has already been generated with this tool, which you should now use for your further analysis.

Figure A.15: Evaluation - Introduction

Introduction to "Code Coverage Density"

The metric "Code Coverage Density" is an extension of classical code coverage which does not only distinguish between "covered" and "uncovered" statements. Particularly, the metric is intended to show how many tests of a given test set cover specific parts of your code. In detail, the metric is defined on 3 abstraction levels which will now be presented.

Line Coverage Density

The following figure shows the test coverage for a method M1. The method consists of 5 lines of code (Line 1 - Line 5). A subset of tests (T1 - T4) of a test suite is considered. The tests cover the lines as depicted below. "0" means that the respective test does not cover the line and "1" means that it does. The coverage density for a line of code is the number of test cases that cover it. Therefore, the line coverage density "LineCovDens" for line 4 would be 3. In relation to the given subset of tests (which contains 4 tests), it holds that 75% of the test cases cover this line.

M1	T1	T2	T3	T4	LineCovDens	CovDens %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%

(a) Line Coverage Density

Method Coverage Density

The following figure shows the test coverage of a method "M1". As in the previous example, there are 4 test cases (T1 - T4) which cover the lines of code as depicted in the table. The coverage density for a method is the arithmetic mean value of the coverage density values for the lines that form the method. In our example, the mean value would be 65%, which means that on average 65% of the tests cover this method.

M1	T1	T2	T3	T4	LineCovDens	CovDens %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%
Mean Density M1					2,6	65,00%

Dispersion Measures

The following figure is an extension of the previous example. In addition to the mean value of the coverage densities of a method's lines of code, we also consider the minimum and maximum coverage densities of the lines. Those values should clearly state how accurate the mean value (and therefore the coverage density of the method M1) actually is.

M1	T1	T2	T3	T4	LineCovDens	CovDens %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%
Mean Density M1					2,6	65,00%
Minimum Density M1					1	25,00%
Maximum Density M1					4	100,00%

(b) Method Coverage Density and Dispersion Measures

Class Coverage Density

The following figure shows the coverage data of two methods M1 and M2, which together form a class "Class A". The coverage density values are already given and computed as explained earlier. The coverage density of a class is (analogous to the computation on method level) the arithmetic mean of the coverage density values of all lines of code, that form that class. In the case below, the coverage density of Class A is the mean of the line coverage densities of Line 1 - Line 12. As for the coverage density on method level, the dispersion measures (minimum and maximum) are also considered here.

Note: the level of abstraction can also be raised to package or module level in analogy to this example.

Class A						
M1	T1	T2	T3	T4	LineCovDens	CovDens %
Line 1	1	1	1	1	4	100,00%
Line 2	1	0	1	1	3	75,00%
Line 3	0	1	1	0	2	50,00%
Line 4	0	1	1	1	3	75,00%
Line 5	0	0	1	0	1	25,00%
Mean Density M1					2,6	65,00%
Minimum Density M1					1	25,00%
Maximum Density M1					4	100,00%
M2	T1	T2	T3	T4	LineCovDens	CovDens %
Line 6	0	1	0	1	2	50,00%
Line 7	1	0	0	1	2	50,00%
Line 8	0	1	1	0	2	50,00%
Line 9	0	1	0	1	2	50,00%
Line 10	0	1	1	0	2	50,00%
Line 11	1	1	1	0	3	75,00%
Line 12	0	1	0	0	1	25,00%
Mean Density M2					2	50,00%
Minimum Density M2					1	25,00%
Maximum Density M2					3	75,00%
Mean Density Class A					2,25	56,25%
Min Density Class A					1	25,00%
Max Density Class A					4	100,00%

(c) Class Coverage Density

Figure A.16: Introduction to Coverage Density Metric

Scenario 1

General Overview

Description

The first step is to get an overall overview concerning the coverage of the software, as well as the distribution of the test cases. For that, you should use the sunburst diagram with the configuration "show all levels". You may configure the color highlighting and the scale factor individually such that you get an optimal overview of the project and the coverage.

Are there any packages or classes, that stand out most, e.g. because of an unbalanced test coverage or their coverage density values?
Q1 - TOPIC B

How evenly are the test cases distributed within the package "at.simianarmy.service" in your opinion?
Q2 - TOPIC B

1 2 3 4 5

not evenly distributed evenly distributed

How well do the data and the visualizations support you for getting a first overview over the test case distribution?
Q3 - TOPIC B

1 2 3 4 5

very poor support very good support

Figure A.17: Evaluation - Scenario 1

Scenario 2

Refactoring

Description

The concrete business logic for the generation of serial letters can be found within the package "at.simianarmy.letter". All relevant classes (e.g. Services, Dataproviders and Worker-Classes) have been implemented here. The Worker-Classes are mainly responsible for the generation of pdf-files, which are transmitted via the REST-interface.

The class "at.simianarmy.letter.worker.SerialLetterWorker" is very complex and unmaintainable and therefore should undergo a refactoring in order to increase the maintainability. You are now asked to give a concrete assessment on this plan based on the data and visualizations given in the report. Note that concrete refactoring steps have not been defined yet (e.g. method splitting etc.).

Code Inspection

The conductor will now present you the visualizations given on package level (i.e. the sunburst diagram for the package "at.simianarmy.letter") and on code level (i.e. the code view for the class "at.simianarmy.letter.worker.SerialLetterWorker").

What is your estimate concerning the code coverage of this class?

Q4 - TOPIC B

1 2 3 4 5

very bad coverage very good coverage

How evenly are the test cases distributed across the class in your opinion?

Q5 - TOPIC B

1 2 3 4 5

not evenly distributed evenly distributed

How well do the data and the visualizations support you concerning the assessment of the risk of a refactoring?

Q6 - TOPIC A

1 2 3 4 5

very poor support very good support

Based on the results shown in the report: is it necessary to write further test cases before a refactoring can be performed in your opinion? If yes, please give concrete proposals where code coverage should be increased (i.e. which lines of code or methods should be assessed by further tests). Please do not only consider lines which are covered or uncovered, but also regard the test case distribution (e.g. reflect the coverage of the method "createPageEventHandler" on line 301).

Q7 - TOPIC B

Figure A.18: Evaluation - Scenario 2

Scenario 3

Test-Levels (Unit, Integration and System Tests)

Description
 For complex data queries, the developers implemented query specification classes that are based on "Spring Specifications" and "Spring Repositories". Those classes can be found within the package "at.simianarmy.service.specification".
 "Spring Specifications" are a framework feature that are used to define queries through query builders. The target is to encapsulate query specific code into specification classes which is then separated from calls to Spring Repositories. The specification classes follow the builder pattern and therefore act like query builders. The conductor will now also show you such a class and its usage in the code view. The clear advantage of this feature is that specification classes can be tested easier than plain queries (e.g. JPQL) within repository classes, as they follow an object oriented implementation style.
 In this scenario, the target is to give an assessment concerning the coverage of the package "at.simianarmy.service.specification" through unit, integration and system tests.
 Note that concerning specification classes, the target is to reach a sufficient coverage through unit tests. Unit tests reside in the same package as the unit under test, whereas integration and system tests may reside in other packages (e.g. at.simianarmy.service, at.simianarmy.repository or at.simianarmy.web.rest)

Conclusion A
 An analysis of the test cases of the class "CompositionSpecification" shows that concrete unit tests for this specification class have already been implemented. There are very detailed test cases within the package "at.simianarmy.service.specification" and the test class "CompositionSpecificationTest". Other classes, e.g. "ClothingSpecification", are not covered through concrete unit tests. Hence, they are covered through integration tests (i.e. tests implemented in the package "at.simianarmy.service" and "at.simianarmy.repository"). The class "InstrumentSpecification" is covered indirectly through tests of the REST endpoint (i.e. through tests implemented in the package "at.simianarmy.web.rest.InstrumentResourceInTest").
 The conclusion is, that the classes "ClothingSpecification" and "InstrumentSpecification" do not have a sufficient test coverage, as the implementation just "gets touched" indirectly through integration tests (in contrast to "CompositionSpecification").

What is your estimate concerning the code coverage of this package? Please do not only consider the coverage on statement level, but also regard different test levels (unit, integration and system tests).
 Q8 - TOPIC B

1 2 3 4 5

very bad coverage very good coverage

How far do you agree with this conclusion?
 Q10 - TOPIC B

1 2 3 4 5

no agreement high agreement

Conclusion B
 The classes "ClothingSpecification" and "InstrumentServiceSpecification" should be covered by separate and independent unit tests.

How evenly are the test cases distributed across the package in your opinion?
 Q9 - TOPIC B

1 2 3 4 5

not evenly distributed evenly distributed

How far do you agree with this conclusion?
 Q11 - TOPIC B

1 2 3 4 5

no agreement high agreement

(a) General Questions and Conclusions

How well do the data and the visualizations support you concerning the discovery of insufficiently covered code parts (i.e. concerning unit, integration and system tests)?
 Q12 - TOPIC B

1 2 3 4 5

very poor support very good support

How well do the data and the visualizations support you concerning the question which types of test cases (unit, integration or system tests) should be added?
 Q13 - TOPIC B

1 2 3 4 5

very poor support very good support

(b) Support Questions

Figure A.19: Evaluation - Scenario 3

Scenario 4

Test Suite Maintenance

Description

Maintainability and Efficiency are two main factors which influence the quality of a test suite. Redundant test cases may affect those factors in a negative way as changes in the code may also mean checking the covering tests and altering them. Furthermore, the test runtime may be unnecessary long if integration or system tests have duplicates.

For this reason, it is also your job to find and eliminate redundant test cases and duplicates.

We are now considering the class "at.simianarmy.service.mapper.qualifier.VerificationLinkQualifier", which provides the implementation for the generation of account activation links for external users. In detail, the class is a spring qualifier, which is used in DTO mapping classes.

Conclusion

The conductor will now lead you through different analysis steps in the sunburst and code view for this package/class.

Following to the sunburst diagram, the class "VerificationLinkQualifier" already has a code coverage of 100% and a well balanced test distribution. As presented in the detail view, the class is covered by the following 4 tests:

- testVerificationLinkForPersonId
- testVerificationLinkForPersonIdWithSSL
- testVerificationLinkForPersonIdWithTenant
- testVerificationLinkForPersonIdWithTenantAndSSL

The naming of the fourth test leads to the premonition that this might be a duplicate of the test cases "testVerificationLinkForPersonWithSSL" and "testVerificationLinkForPersonWithTenant". Therefore, we will have a closer look at the code view and analyze how those tests cover the code exactly.

Through adding and hiding the test cases, we reveal that 100% statement coverage is already reached through the following tests:

- testVerificationLinkForPersonId
- testVerificationLinkForPersonIdWithSSL
- testVerificationLinkForPersonIdWithTenant

The test case "testVerificationLinkForPersonWithTenantAndSSL" should be further analyzed concerning test redundancy, as the lines of code of this class are already covered by other tests that cover this class.

How far do you agree with this conclusion?
Q14 - TOPIC C

1 2 3 4 5

no agreement high agreement

How well do the data and the visualizations support you concerning the search and discovery of test redundancies and duplicates?
Q15 - TOPIC C

1 2 3 4 5

very poor support very good support

Figure A.20: Evaluation - Scenario 4

A.3 Listings

```

{
  "projectName": "EXAMPLE 1.0-SNAPSHOT",
  "coverageMetrics": {
    "overallCoverage": ...,
    "methodCoverage": ...,
    "statementCoverage": ...,
    "branchCoverage": ...
  },
  "packages": {...},
  "testCases": {
    "example.ExampleClassTest.testA": {
      "className": "ExampleClassTest",
      "name": "testA",
      "packageName": "example",
      "testSuccess": true,
      "lineNumber": 10,
      "classFilePath": "\\input-project\\src\\test\\java\\example\\ExampleClassTest.java",
      "abstract": false
    },
    "example.ExampleClassTest.testB": {
      "className": "ExampleClassTest",
      "name": "testB",
      "packageName": "example",
      "testSuccess": true,
      "lineNumber": 20,
      "classFilePath": "\\input-project\\src\\test\\java\\example\\ExampleClassTest.java",
      "abstract": false
    }
  },
  "numTestCases": 2,
  "coveringTests": [
    "example.ExampleClassTest.testA",
    "example.ExampleClassTest.testB"
  ],
  "numCoveringTests": 2,
  "size": 20
}

```

Listing A.1: JSON Structure (Top Level)

```

{
  ...
  "packages": {
    "example": {
      "packageName": "example",
      "qualifiedName": "example",
      "size": 20,
      "coverageMetrics": {...},
      "packages": {},
      "coveringTests": [
        "example.ExampleClassTest.testA",
        "example.ExampleClassTest.testB"
      ],
      "numCoveringTests": 2,
      "classes": {
        "ExampleClass": {
          "className": "ExampleClass",
          "qualifiedName": "example.ExampleClass",
          "size": 20,
          "coverageMetrics": {...},
          "methods": {
            "ExampleClass": {...},
            "methodA": {...},
            "methodB": {
              "lineNumber": 10,
              "methodName": "methodB",
              "qualifiedName": "example.ExampleClass.methodB",
              "lines": {
                "11": {
                  "lineNumber": 11,
                  "lineSpan": 1,
                  "coveringTests": ["example.ExampleClassTest.testB"],
                  "numCoveringTests": 1
                },
                "12": {
                  "lineNumber": 12,
                  "lineSpan": 1,
                  "coveringTests": ["example.ExampleClassTest.testB"],
                  "numCoveringTests": 1
                }
              }
            },
            "coveringTests": [],
            "numCoveringTests": 0
          }
        },
        "coveringTests": [...],
        "numCoveringTests": 2
      }
    }
  }
}

```

Listing A.2: JSON Structure (Class Level)



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

1.1	Methodology outline	3
2.1	Software Product Quality [3]	8
2.2	Errors, faults and failures [6]	12
2.3	Comparison of black-box and white-box testing strategies [6]	14
2.4	Control flow graph example [6]	15
2.5	Refactoring: Abstract Example [19]	16
3.1	<i>JaCoCo</i> HTML Report: Coverage Information	25
3.2	<i>JaCoCo</i> HTML Report: Class View	25
3.3	<i>Cobertura</i> Coverage Report	26
3.4	<i>Cobertura</i> Coverage Report: Class View	26
3.5	Correlation View in <i>CodeCover</i> [55]	27
3.6	Code Highlighting and Hot Path in <i>CodeCover</i> [55]	28
3.7	<i>OpenClover</i> Top Risks	29
3.8	<i>OpenClover</i> Quick Wins	29
3.9	<i>OpenClover</i> Coverage Tree Map	30
3.10	<i>OpenClover</i> Class View	30
5.1	Realization Proposals: Sunburst Diagram	40
5.2	Realization Proposals: Sunburst Details	41
5.3	Realization Proposals: Class View	43
5.4	Realization Proposals: Class View Details	43
5.5	Realization Proposals: Test Set Filter	44
5.6	Realization Proposals: Radar Chart	45
5.7	Realization Proposals: Bar Chart	46
5.8	Realization Proposals: Radar Chart Details	46
5.9	Visualization Proposals: Test Details	47
5.10	Metric relevance ratings (1 = <i>not relevant</i> , 5 = <i>highly relevant</i>)	50
5.11	Feature/Mockup relevance ratings (1 = <i>not relevant</i> , 5 = <i>highly relevant</i>)	52
5.12	Test Details with Test Code	56
6.1	Prototype Execution Phases	60
6.2	Example report directory structure	63
6.3	Prototype Execution Automation	64

6.4	Coverage Density Report: Sunburst Diagrams	66
6.5	Coverage Density Report: Sunburst Diagram Configurations	67
6.6	Coverage Density Report: Sunburst View Details	68
6.7	Coverage Density Report: Code View	69
6.8	Coverage Density Report: Test Selection	70
6.9	Coverage Density Report: Test View	71
7.1	Method “createPageEventHandler” in class view for “SerialLetterWorker”	78
7.2	Sunburst view for package “at.simianarmy.specification”	79
7.3	Example specification class (code snippet)	80
7.4	Details for various specification classes	82
7.5	Class view for the class “VerificationLinkQualifier”	83
7.6	Detail view for the class “VerificationLinkQualifier”	84
7.7	Answers for question <i>Q2</i> in scenario 1	86
7.8	Answers for question <i>Q3</i> in scenario 1	86
7.9	Answers for question <i>Q4</i> in scenario 2	88
7.10	Answers for question <i>Q5</i> in scenario 2	88
7.11	Answers for question <i>Q6</i> in scenario 2	89
7.12	Answers for question <i>Q8</i> in scenario 3	89
7.13	Answers for question <i>Q9</i> in scenario 3	90
7.14	Agreements on conclusion A in scenario 3	90
7.15	Agreements on conclusion B in scenario 3	90
7.16	Answers for question <i>Q12</i> in scenario 3	91
7.17	Answers for question <i>Q13</i> in scenario 3	91
7.18	Agreements on conclusion C in scenario 4	92
7.19	Answers for question <i>Q15</i> in scenario 4	92
7.20	Rating per <i>support question</i> (1 = <i>very poor support</i> , 5 = <i>very good support</i>)	93
7.21	Improvement Suggestion: Color Gradients for Class Sections	94
7.22	Rating per research question (1 = <i>very poor support</i> , 5 = <i>very good support</i>)	97
A.1	Expert Interviews - General Demographic Questions	105
A.2	Metric evaluation questions (1)	106
A.3	Metric evaluation questions (2)	107
A.4	Metric evaluation questions (3)	108
A.5	Mockup evaluation questions - Sunburst (1)	109
A.6	Mockup evaluation questions - Sunburst (2)	110
A.7	Mockup evaluation questions - Class View (1)	111
A.8	Mockup evaluation questions - Class View (2)	112
A.9	Mockup evaluation questions - Class View (3)	113
A.10	Mockup evaluation questions - Radar Chart (1)	114
A.11	Mockup evaluation questions - Radar Chart (2)	115
A.12	Mockup evaluation questions - Bar Chart	116
A.13	Mockup evaluation questions - Test Details	117
A.14	Evaluation - General Demographic Questions	118

A.15 Evaluation - Introduction	118
A.16 Introduction to Coverage Density Metric	119
A.17 Evaluation - Scenario 1	120
A.18 Evaluation - Scenario 2	121
A.19 Evaluation - Scenario 3	122
A.20 Evaluation - Scenario 4	123



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1	LineCovDens example	36
4.2	MethodCovDens example	37
4.3	ClassCovDens example: computation of LineCovDens values	38
4.4	ClassCovDens example: values for class C_1	38
5.1	Final Requirements	57

Listings

6.1	clover-executor.sh	65
A.1	JSON Structure (Top Level)	124
A.2	JSON Structure (Class Level)	125



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

- %ClassCovDens** Relative Class Coverage Density. 37, 38
- %LineCovDens** Relative Line Coverage Density. 36
- %MethodCovDens** Relative Method Coverage Density. 36, 37
- AI** Artificial Intelligence. 31
- CBO** Coupling between Object. 10
- CI** Continuous Integration. 22, 31
- ClassCovDens** Class Coverage Density. 37–40, 48, 49, 94, 131
- DIT** Depth of Inheritance Tree. 10
- DOM** Document Object Model. 61, 63
- GUI** graphical user interface. 54
- IDE** Integrated Development Environment. 20, 25, 27, 31, 54, 55
- IEEE** Institute of Electrical and Electronic Engineers. 7, 9
- ISO** International Organization for Standardization. 7
- LCOM** Lack of Cohesion in Methods. 10
- LineCovDens** Line Coverage Density. 35, 36, 38, 42, 44, 48–50, 96, 131
- LOC** Line of Code. 1, 2, 10, 25, 26, 30, 31, 33, 35–38, 42, 49–51, 54, 55, 60, 62, 63, 69, 74, 75, 84, 87, 88, 93, 94, 96
- MethodCovDens** Method Coverage Density. 36–38, 45, 48, 49, 53, 131
- MTTF** Mean Time to Failure. 11

NOS	Number of Statements.	10
PIVoT	Project Insights and Visualization Toolkit.	21
SPA	Single Page Application.	75
SPI	Software Process Improvement.	10
SQA	Software Quality Assurance.	9–11
SQC	Software Quality Control.	9
SQM	Software Quality Management.	9
SQP	Software Quality Planning.	9
SUT	System under Test.	24

References

- [1] Y. Singh, *Software Testing*. Cambridge University Press, 2012, ISBN: 9781139196185.
- [2] *ISO 9000:2015. Quality management systems - Fundamentals and vocabulary*, 4th ed. Geneva, Switzerland, 2015.
- [3] *ISO/IEC 25010:2011. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*, 1st ed. Geneva, Switzerland, 2011.
- [4] *IEEE Standard for Software Quality Assurance Processes*. 2014, ISBN: 9780738191683.
- [5] P. Rechenberg and G. Pomberger, *Informatik-Handbuch*, 4th ed. Hanser, 2006, ISBN: 3446401857.
- [6] A. Schatten, M. Demolsky, D. Winkler, S. Biffel, E. Gostischa-Franta, T. Östreicher, and A. Schatten, *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. Heidelberg: Spektrum Akademischer Verlag, 2010, ISBN: 9783827424877.
- [7] S. Wagner, *Software Product Quality Control*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN: 9783642385711.
- [8] T. Grechenig, M. Bernhart, R. Breiteneder, and K. Kappel, *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, 2010, ISBN: 9783868940077.
- [9] *Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK Guide V3.0)*. 2014, ISBN: 9780769551661.
- [10] *IEEE Standard For A Software Quality Metrics Methodology Revision And Reaffirmation*. 2005, ISBN: 1559375299.
- [11] N. E. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*, 3rd ed., ser. Chapman & Hall/CRC Innovations in Software Engineering and Software Development. Boca Raton: CRC Press, 2015, ISBN: 042910622X.
- [12] A. Abran, *Software Metrics and Software Metrology*, 1st ed. Hoboken: Wiley, 2010, ISBN: 0470597208.
- [13] S. H. Kan, *Metrics and models in software quality engineering*, 2nd ed. Boston: Addison-Wesley, 2003, ISBN: 0201729156.

- [14] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*, 3rd ed. Hoboken, N.J.: John Wiley & Sons, 2012, ISBN: 1118133137.
- [15] A. Spillner, T. Linz, H. Schaefer, M. Barabas, J. Flynn, and H. Kraus, *Software testing foundations : a study guide for the certified tester exam : foundation level, ISTQB compliant*, 4th ed. Santa Barbara, California: Rocky Nook, 2014, ISBN: 1306807603.
- [16] P. C. Jorgensen, *Software Testing: A Craftman's Approach*. Auerbach Publications, 2018, ISBN: 1466560681.
- [17] "IEEE/ISO/IEC International Standard - Software and systems engineering–Software testing–Part 4: Test techniques," *ISO/IEC/IEEE 29119-4:2021(E)*, pp. 1–148, 2021.
- [18] M. Fowler, *Refactoring: Improving the Design of Existing Code*, eng, 2nd ed. Addison-Wesley Professional, 2018, ISBN: 9780134757681.
- [19] M. Lemaire, *Refactoring at Scale*. Sebastopol: O'Reilly Media, Incorporated, 2020, ISBN: 9781492075530.
- [20] J. Kerievsky, *Refactoring to patterns*, 1st ed., ser. Addison-Wesley signature series. Boston: Addison-Wesley, 2005, ISBN: 0321630017.
- [21] D. Bowes, T. Hall, J. Petrić, T. Shippey, and B. Turhan, "How Good Are My Tests?" *International Workshop on Emerging Trends in Software Metrics, WETSoM*, pp. 9–14, 2017, ISSN: 23270969.
- [22] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, 2014, pp. 12–23, ISBN: 9781450327565.
- [23] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*, 2012, ISBN: 9781450316149.
- [24] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, New York, NY, USA: ACM, 2012, 51:1–51:11, ISBN: 9781450316149.
- [25] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings - International Conference on Software Engineering*, vol. 1, 2010, pp. 175–184, ISBN: 9781605587196.
- [26] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 344–353.
- [27] V. S. Sharma, R. Mehra, and V. Kaulgud, "What do developers want? An advisor approach for developer priorities," in *Proceedings - 2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2017*, IEEE, 2017, pp. 78–81, ISBN: 9781538640395.

- [28] V. S. Sharma and V. Kaulgud, “PIVoT: Project insights and Visualization Toolkit,” in *2012 3rd International Workshop on Emerging Trends in Software Metrics, WETSoM 2012 - Proceedings*, IEEE, 2012, pp. 63–69, ISBN: 9781467317627.
- [29] J. Biehl, M. Czerwinski, G. Smith, G. Robertson, and B. Bailey, “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams,” in *CHI 2007 Conference on Human Factors in Computing Systems*, San Jose, California, USA: Association for Computing Machinery, 2007, pp. 1313–1322, ISBN: 9781595935939.
- [30] A. Sarma, Z. Noroozi, and A. van der Hoek, “Palantír: Raising Awareness among Configuration Management Workspaces,” in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03, USA: IEEE Computer Society, 2003, pp. 444–454, ISBN: 076951877X.
- [31] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. Los Alamitos, California: IEEE Computer Society Press, 1996, ISBN: 9780818673849.
- [32] X. Ren, F. Shah, B. Ryder, O. Chesley, and J. Dolby, “Chianti: A Prototype Change Impact Analysis Tool for Java,” Rutgers University Department of Computer Science, Tech. Rep., 2003.
- [33] S. Amann and E. Jürgens, “Change-Driven Testing,” in *The Future of Software Quality Assurance*, S. Goericke, Ed., Cham: Springer International Publishing, 2020, ISBN: 978-3-030-29509-7.
- [34] J. Wloka, B. G. Ryder, and F. Tip, “JUnitMX - A change-aware unit testing tool,” in *Proceedings - International Conference on Software Engineering*, 2009, pp. 567–570, ISBN: 9781424434527.
- [35] X. Ren, B. G. B. Ryder, M. Stoerzer, F. Tip, F. Shah, F. Tip, B. G. B. Ryder, and O. Chesley, “Chianti: A Tool for Change Impact Analysis of Java Programs,” in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '04, New York, NY, USA: Association for Computing Machinery, 2004, pp. 432–448, ISBN: 1581138318.
- [36] A. van Deursen, L. Moonen, A. van Den Bergh, and G. Kok, “Refactoring test code,” *Extreme Programming Perspectives*, no. November, pp. 141–152, 2002.
- [37] N. Koochakzadeh, V. Garousi, and F. Maurer, “Test redundancy measurement based on coverage information: Evaluations and lessons learned,” *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009*, pp. 220–229, 2009.
- [38] M. J. Harrold, R. Gupta, and M. L. Soffa, “A Methodology for Controlling the Size of a Test Suite,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 270–285, 1993, ISSN: 15577392.
- [39] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, “An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites,” *Conference on Software Maintenance*, pp. 34–43, 1998.

- [40] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990, ISBN: 0716710455.
- [41] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, vol. 2, pp. 1–16, 1995.
- [42] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE International Conference on Software Maintenance, ICSM*, vol. 29, no. 3, pp. 92–103, 2001.
- [43] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for Reducing the Size of Coverage-based Test Sets," *International Conference on Testing Computer Software*, pp. 1–11, 1995.
- [44] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4422 LNCS, no. March, pp. 291–305, 2007, ISSN: 16113349.
- [45] L. Inozemtseva and R. Holmes, "Coverage is Not Strongly Correlated with Test Suite Effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, New York, NY, USA: Association for Computing Machinery, 2014, pp. 435–445, ISBN: 9781450327565.
- [46] Y. Wei, B. Meyer, and M. Oriol, "Is branch coverage a good measure of testing effectiveness?" *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7007 LNCS, pp. 194–212, 2011, ISSN: 03029743.
- [47] D. Tengeri, Á. Beszédes, T. Gergely, L. Vidács, D. Hávas, and T. Gyimóthy, "Beyond code coverage - An approach for test suite assessment and improvement," *2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2015 - Proceedings*, no. April, 2015.
- [48] R. P. Buse and T. Zimmermann, "Information needs for software development analytics," in *Proceedings - International Conference on Software Engineering*, ser. ICSE 2012, Piscataway, NJ, USA: IEEE Press, 2012, pp. 987–996, ISBN: 9781467310673.
- [49] J. F. Smart, *Jenkins*. Sebastopol: O'Reilly Media, Incorporated, 2011, ISBN: 9781449305352.
- [50] K. Yilmaz, "Comparison of Quantitative and Qualitative Research Traditions: epistemological, theoretical, and methodological differences," eng, *European journal of education*, vol. 48, no. 2, pp. 311–325, 2013, ISSN: 0141-8211.

Online References

- [51] E. Dietrich, *A Better Metric than Code Coverage*, [Online; visited on 01/04/2021]. [Online]. Available: <https://daedtech.com/a-better-metric-than-code-coverage/>.
- [52] Parasoft, *Code Coverage Density and Test Overlap*, [Online; visited on 01/04/2021]. [Online]. Available: <https://www.parasoft.com/code-coverage-density-and-test-overlap/>.
- [53] JaCoCo, *Coverage Counters*, [Online; visited on 01/05/2022]. [Online]. Available: <https://www.eclemma.org/jacoco/trunk/doc/counters.html>.
- [54] A. Altvater, *The Ultimate List of Code Coverage Tools: 25 Code Coverage Tools for C, C++, Java, .NET, and More*, [Online; visited on 01/05/2022]. [Online]. Available: <https://stackify.com/code-coverage-tools/>.
- [55] CodeCover, *HOWTO Use CodeCover: From code to report, a complete walkthrough*, [Online; visited on 01/05/2022]. [Online]. Available: http://codecover.org/documentation/tutorials/how_to_complete.html.
- [56] Parasoft, *Parasoft Jtest Capabilities*, [Online; visited on 01/14/2022]. [Online]. Available: <https://www.parasoft.com/products/parasoft-jtest/>.
- [57] Parasoft, *Parasoft Jtest: Data Sheet*, [Online; visited on 01/14/2022]. [Online]. Available: <https://www.parasoft.com/wp-content/uploads/2021/02/datasheet-jtest.pdf>.
- [58] Atlassian, *Configuring a job's build artifacts*, [Online; visited on 10/27/2021]. [Online]. Available: <https://confluence.atlassian.com/bamboo/configuring-a-job-s-build-artifacts-289277071.html>.