A Method for Comparing and Selecting Static Code Analysis Tools in Software Development Projects

Christian Haertel (christian.haertel@ovgu.de)
Otto-von-Guericke-University
Matthias Pohl
Otto-von-Guericke-University
Daniel Staegemann
Otto-von-Guericke-University
Klaus Turowski
Otto-von-Guericke-University

Research Article

Keywords: Deployment Pipeline, Software Tests, Static Code Analysis Tools, Software Quality Models, ISO 25010, Fuzzy TOPSIS

Posted Date: April 18th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1546855/v1

A Method for Comparing and Selecting Static Code Analysis Tools in Software Development Projects

Christian Haertel[1*], Matthias Pohl[1†], Daniel Staegemann[1†] and Klaus Turowski[1†]

[1*]Institute of Technical and Business Information Systems, Otto-von-Guericke-University, Universitaetsplatz 12, Magdeburg, 39104, Saxony-Anhalt, Germany.

*Corresponding author(s). E-mail(s): christian.haertel@ovgu.de;
Contributing authors: matthias.pohl@ovgu.de;
daniel.staegemann@ovgu.de; klaus.turowski@ovgu.de;
†These authors contributed equally to this work.

Abstract

To enable reliable software releases, automated concepts are increasingly being sought as part of the necessary test processes to be able to detect potential errors at an early stage. Static code analysis tools (SCATs) are especially suited for this purpose, as these testing tools perform their checks without actually executing the software. Thus, they represent an important part in the test suite. For the problem of carefully selecting such a tool for project use, a method for the construction of SCAT comparison catalogs is developed in this article to allow to contrast different tools and to evaluate them with respect to derived criteria. While the comparison categories are derived from established software quality models by taking into account the specifics of SCATs, multi criteria decision making procedures employing linguistic predicates are used to determine the most suitable test tool for each case. The artifact is demonstrated and evaluated in an artificial SCAT selection process which involves FindBugs, checkstyle, and PMD.

Finally, potential extensions of the proposed method are outlined and its simple modifiability to other software decision situations is set forth.

Keywords: Deployment Pipeline, Software Tests, Static Code Analysis Tools, Software Quality Models, ISO 25010, Fuzzy TOPSIS

1 Introduction

Software releases frequently represent risky and time-consuming projects (Humble and Farley, 2011). Therefore, Software engineering methods such as Continuous Delivery (CDE), Continuous Integration (CI) or Continuous Engineering are increasingly being used to counteract vulnerabilities along the Software Development Life Cycle (SDLC) (Lunn, 2003) and to enable companies to develop their own software securely and efficiently. Although the demarcation between these individual concepts does not always appear to be clear, the basic principle of these approaches can be summarized as follows. In general, the development cycles should be kept incremental and the software should always be in a working state (Chen, 2015). This enables a reliable release in conjunction with a reduced feedback loop. In order to effectively perform such continuous practices, automated processes are sought along the entire deployment or integration pipeline (Humble and Farley, 2011). Therefore, the establishment of the presented concepts requires considerable organizational efforts.

In the SDLC, these endeavors are especially highlighted with the point testing. This term includes countless buzzwords that must all be considered in a sofware project: Testing against functional as well as non-functional requirements, unit tests, acceptance tests, component tests, integration testing, showcases, exploratory testing, usability tests etc. This plethora of terms is indicative of the effort that software testing teams must expend to ensure

that the final product is deployed in a satisfactory state. Consequently, supporting developers and testers with automated processes is inevitable. While automating all aspects is not necessarily feasible (Marick, 1998), an automated test suite can map a large portion of the required checks. This usually includes static code analysis tools (SCATs). These tools enable frequent as well as quick checks before the software is executed (Novak et al, 2010). Consequently, development teams are confronted with the question which of these checking tools, available from various vendors, are most suitable for the integration into the test suite to design the testing process in accordance with the principles of CDE/CI. Thus, to establish simplicity and transparency, this software selection problem requires an approach that allows comparison and decision between various of such analysis tools for a specific use case. While there are some papers (e. g. Arusoaie et al (2017) and Kaur and Nayyar (2020)) which contrast sets of SCATs for a certain project, the portability of these results to other tool sets or other project circumstances remains questionable. Furthermore, a decision-making procedure that facilitates the simplicity and reproducibility of the SCAT selection is often missing. Therefore, the chapter at hand proposes an ubiquitous method for the comparison of SCAT that allows to take into the consideration the circumstances of the respective situation.

The contribution at hand utilizes the Design Science Research (DSR) method (Hevner et al, 2004). The detailed methodological approach, including the introduction of the research questions, is described in the upcoming paragraph. Section 3 covers the necessary theoretical basics for the artifact by discussing the terms Continuous Software Engineering, Continuous Integration, Continuous Delivery as well as Continuous Deployment, introducing the Deployment Pipeline, explaining the different concepts of software testing, and examining static code analysis tools in detail. Section 4 presents the SCAT

selection method constructed on the theoretical findings. Subsequently, the fundamental functionality of the artifact is demonstrated in Section 5. With the help of the method, a SCAT evaluation catalog using adapted criteria from the ISO 25010 standard and the Fuzzy TOPSIS decision making procedure is derived. This instantiation of the artifact is applied in an artificial SCAT selection situation between three test tools. Based on the demonstration, the evaluation of the method is undertaken in Section 6. The chapter concludes with a summary of the results regarding the research questions, compiles the limitations, and gives an outlook on potential extensions and improvements.

2 Methodology and Research Objectives

As indicated in the introduction, the aim of this study is the proposition of a method for the development of a comparison catalog for SCATs to aid with selecting the appropriate tool for the respective software project. Therefore, the following research questions (RQs) are answered:

RQ 1: What are the application fields of SCATs in (modern) software testing with regards to their capabilities, strengths and weaknesses?

RQ 2: Considering the findings from RQ 1, what should be the design of a systematic SCAT selection process, also taking into account the specific circumstances of the respective software development project?

This work follows the Design Science Research methodology to derive the described artifact and engage in the stated RQs (Hevner et al, 2004). This section is dedicated to the detailed explanation of the methodological approach that is used in this contribution. Regarding the DSR, this article specifically follows the process prescribed by Peffers et al (2007), consisting

of the stages problem identification and motivation, objectives of a solution, design and development, demonstration, evaluation and communication.

Problem identification and motivation. The introduction already highlighted the problem in some capacity. Software releases are risky and require comprehensive testing. As most projects are too complex and therefore cannot afford to rely solely on manual code review, static code analysis tools became increasingly relevant. However, to the best of our knowledge, there is currently no systematic, transparent and easy-to-use method for comparing and selecting SCATs for a specific software project. While there are some articles contrasting different sets of SCATs for a given use case (e. g. security) (e. g. Chatzieleftheriou and Katsaros (2011) and Kaur and Nayyar (2020)), these evaluations are not designed for different backgrounds since they do not consider the specific requirements of the software development project. Accordingly, a methodological and ubiquitously applicable approach that allows evaluating and choosing a SCAT for a given project needs to be constructed.

Objectives of a solution. Based on the problem definition, the general goal of the proposed artifact is to strengthen the testing process in software projects and thus, the quality of the outcome. More specifically, the SCAT evaluation method intends to enrich this selection process in terms of systematics, transparency, usability and speed. Decision makers in software projects shall be enabled to derive a custom SCAT catalog that reflects the circumstances of the respective undertaking.

Design and development. The construction of the artifact is lead by the initially posed research questions. Answering RQ 1 provides the necessary background and certain specifics for the SCATs that are required to be able

to properly contrast the tools. Accordingly, the proposed SCAT evaluation method builds on this foundation in conjunction with software quality models to develop comparison criteria that reflect the SCAT theory and project specifics. In the second step, the systematic and transparent evaluation of SCATs according to the criteria implies the application of a decision making method to achieve a so-called winning alternative.

Demonstration. The demonstration's goal is to show the effectiveness of the artifact to solve the outlined problem (Peffers et al, 2007). Therefore, the proposed method is applied to create a SCAT comparison catalog, using the criteria of the ISO 25010 standard for software quality as a basis and Fuzzy TOPSIS as the decision making procedure. Then, a SCAT selection process is simulated by evaluating three SCATs according to the catalog.

Evaluation. Based on the results of the preceding demonstration, this step analyzes how the artifact contributes to a solution of the initially stated problems: improving the test process in software projects and to increase, among others, transparency, simplicity and efficiency in the SCAT selection process with the proposed method.

Communication. The communication is achieved through the documentation and publication of our studies.

The key aspects of the application of the DSR process in the context of this article is also graphically summarized with the help of the DSR grid (see Figure 1) (vom Brocke and Maedche, 2019).

3 Prerequisites

The Design and Development stage of the DSR process in this work involves answering RQ 1. In order to determine the characteristics of SCATs that help

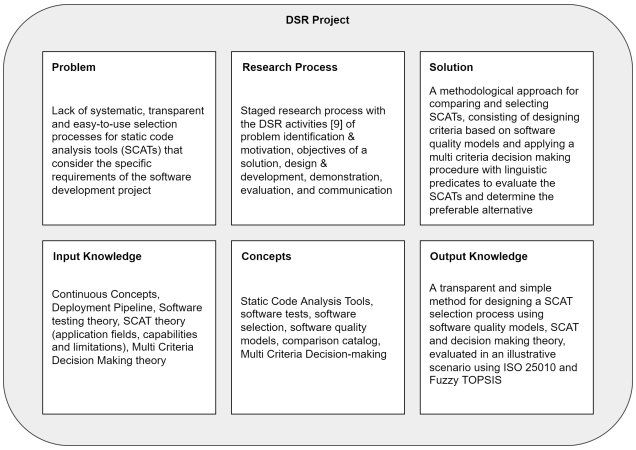

Fig. 1 DSR Grid for this project, according to vom Brocke and Maedche (2019)

with developing the artifact, a deep dive into the respective theory of these tools and (modern) software testing in general is needed.

3.1 Distinction of the Continuous Concepts

First of all, the Continuous concepts of software engineering need to be differentiated from each other. The four mentioned terms are Continuous Software Engineering, Continuous Integration, Continuous Delivery and Continuous Deployment. The common aim persists in reducing the number of time-consuming and error-prone software releases for companies (Humble and Farley, 2011). Under the keyword automation, organizations shall be enabled to publish new features and software products in increased frequency and reliability (Shahin et al, 2017). As more and faster feedback can be gained from the software development cycle and the users, this aspect promises a higher customer satisfaction and product quality. However, it is pointed out that an appropriate application of the mentioned practices requires a high organizational effort (Shahin et al, 2017)(Humble and Farley, 2011).

Continuous Software Engineering (CSE) comprises the development and deployment of software including a fast retrieval of feedback from the program

and from the user (Bosch, 2014)(Fitzgerald and Stol, 2017). The whole process consists of three phases: Business Strategy and Planning, Development and Operations. Continuous Integration, Continuous Delivery and Continuous Deployment belong to the practices of the development phase. Therefore, CSE is positioned at a higher-level in relation to the other concepts.

Continuous Integration can be used to ensure that development teams achieve a higher degree of control in their collaboration within the creation of large and complex systems (Humble and Farley, 2011). With this procedure, the developers integrate their work packages (e. g. code) regularly (several times a day) to comply with shorter and more frequent release cycles (Fitzgerald and Stol, 2017). This is expected to improve the final quality and team productivity through the provision of quick feedback on new problems with each code change (Humble and Farley, 2011) and changes in smaller increments which are easier to debug (Meyer, 2014). This is also harnessed by other quality assurance strategies such as test driven development, where CI is often utilized (Staegemann et al, 2021). For this purpose, CI often includes automated software building and testing (Leppänen et al, 2015).

In contrast to Shahin et al (2017), other papers employ the terms Continuous Delivery and Continuous Deployment (CD) synonymously. The two terms mainly differ in the type of the production environment where the software is initially deployed at. Both practices aim to offer faster and more reliable releases to users and customers (Shahin et al, 2017). CDE is supposed to guarantee that an application is always in an executable state, following the successful completion of a series of automatic tests and quality checks (Weber et al, 2016). Subsequently, the software is automatically delivered to a production-like environment to lower the risks during the final deployment

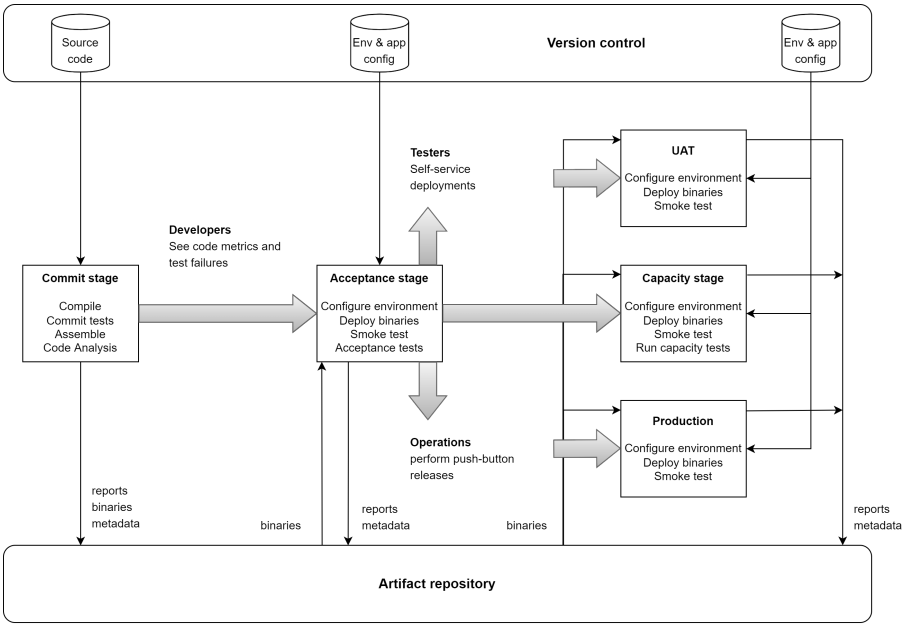

Fig. 2 Example for a Deployment Pipeline according to Humble and Farley (2011)

by gained user feedback. Due to this fact, it becomes clear that the previously described CI is considered essential for efficient CDE (Shahin et al, 2017).

CD extends this and delivers the application automatically and continuously (i. e. with every change) to the respective production or customer environment (Weber et al, 2016). The application of CDE always implies the presence of CD, as otherwise the release to the end user would be missing (Shahin et al, 2017). Conversely, CD without CDE remains possible. The deployment of changes to the production environment is accomplished via the "Deployment Pipeline", which will be discussed in the following subsection.

3.2 Deployment Pipeline and Testing

For the context of this article, it is important to examine a basic deployment pipeline to understand the role of the static code analysis tools in this process. First of all, following the definitions discussed in Section 3.1, the question

arises why CD is required when a large number of software tests was already conducted in the CI scope and only the roll-out to a production environment is remaining. In particular, the stages leading to the user environment do not represent a trivial process at all, since most problems occur during the progress of the software through tests up to the actual operation (Humble and Farley, 2011). CI is mainly limited to the development team: the code must compile successfully and pass a series of unit and acceptance tests to ensure the application is always kept in an executable state. The other teams for testing and operations are virtually not included which increases the probability of bugs because the feedback cycle between the divisions takes too long (Humble and Farley, 2011). If the project does not manage to use the program in a production-like environment in a timely manner, the software becomes "undeployable".

Therefore, CD offers the possibility to strengthen the connection between the development and operations team (Shahin et al, 2017) because a Deployment Pipeline induces a certain degree of collaboration in this respect (Humble and Farley, 2011). According to the definition in Humble and Farley (2011) (Humble and Farley, 2011), the purpose of the Deployment Pipeline lies in the creation, testing and commissioning of complex systems with a higher quality and lower costs. It represents an automated manifestation of the process from version control up to the user of the software. Thus, CI provides the input for this part of the release sequence (Humble and Farley, 2011). Within the scope of the deployment pipeline, each change passes a complex procedure until final commissioning. This includes building, various test stages and finally the commissioning in the production environment. A visualization of this process is illustrated in Figure 2 and is explained in more detail below.

The Deployment Pipeline consists of several stages and starts with the developers bringing in changes into their version control system (Humble and Farley, 2011). The newly created instance must first traverse the commit stage where it is ensured that the system works on a technical level. This includes the compilation of the code, a series of unit tests and code analysis. If these checks are passed successfully, the instance is stored in the artifact repository. Subsequently, the automated acceptance tests of the second stage can be triggered, since it must be verified whether the application works correctly on both the functional as well as the non-functional layer and the requirements of the customers are met. In the further course of the pipeline, the process splits into three branches, reflecting the deployment in different environments for additional testing purposes (Humble and Farley, 2011). In contrast to the previous steps, the transition into these environments is ideally initiated manually by the testers or the operations team.

In summary, user acceptance tests (UATs) are intended to assure the usability of the system. Moreover, they search for errors that the automated tests did not reveal (Humble and Farley, 2011). After the capacity tests, just the release phase remains. The operations team pushes the application into a production environment to provide it to the user. The goal is to obtain as much feedback as possible regarding the behavior and state of the software in each version within a short time frame to save time and money on debugging.

3.3 An Overview of Software Testing

The presentation of the Deployment Pipeline provided a basic impression of the respective stages without technical details. Various types of software tests were mentioned but initially, no adequate clarification of the terms was given. Therefore, the goal of this subsection is to gain an overview of the large amount

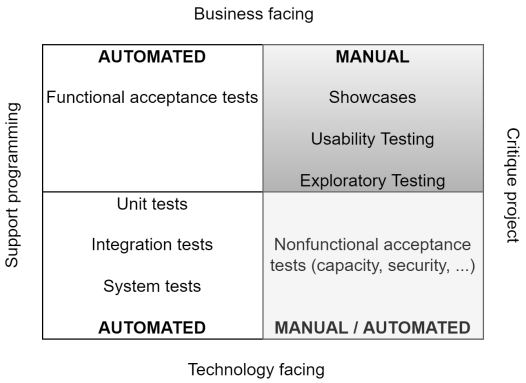

Fig. 3 Testing quadrants according to Humble and Farley (2011)

of used tests to allow covering static code analysis tools according to the aim of this chapter.

Since plenty of checks are performed in the software development cycle, a categorization among the types of tests is desirable. Brian Marick introduced such a concept in 2003 (Humble and Farley, 2011), Crispin and Gregory (2009) picked up on it later and increased its recognition. As can be understood in Figure 3, software tests in this model are classified depending on whether they are business-facing or technology-facing and whether they support the development process or critique the project (Humble and Farley, 2011). This differentiation results in the four quadrants which are displayed in the graphic.

The first quadrant describes the business-facing test that support the development process. This refers to functional acceptance tests, which were already mentioned in the last section. Such tests verify the satisfaction of the acceptance criteria of a user story by the system (Abran et al, 2004). Preferably, these are automated and written before the development of a story. Acceptance tests are utilized to evaluate several attributes of the designed system. Examples contain functionality, capacity, usability, security, modifiability and availability. Only checks affecting the functionality belong to this category. The non-functional acceptance tests are instead assigned to the fourth quadrant,

although the boundaries do not always appear clear. Generally, acceptance tests are considered indispensable as they help developers to understand when their work is sufficient. Furthermore, they show users to which extent the application corresponds to their own ideas (Humble and Farley, 2011).

Technology-facing tests supporting the development process are also automated and exclusively written and maintained by developers (Humble and Farley, 2011). Unit tests review the functioning in isolation of software pieces which are separately testable, using so-called test doubles (Abran et al, 2004). These tests usually run fast and provide early feedback since they generally do not cover the interaction between system components (e. g. with databases, file systems or external systems). Consequently, unit tests disregard bugs that are associated with the interaction of different entities of the application. Component and integration tests are meant to fill this gap. Sometimes, these two terms are used synonymously. Humble and Farley (2011) specify the two terms as follows: Component tests focus on larger functionality clusters and thus are more time-consuming, because the interaction is considered at this stage (Humble and Farley, 2011). On the other hand, integration tests refer to checks ensuring that each independent module of the application is working properly with the respective services it requires. Deployment or installation tests are performed when commissioning the software in the target environment (Abran et al, 2004). It is evaluated that the application is correctly installed, configured, is able to interact with the required services and responds (Humble and Farley, 2011).

In contrast to the previous two quadrants, business-facing tests critiquing the project contain manual checks. The intention of these tests is to confirm that the program is indeed providing the expected value to the user. This involves less the compliance with the formulated specifications, but rather

the verification that the requirements themselves are accurate (Humble and Farley, 2011). As software is never specified perfectly in advance, great importance is attached to such examinations. Showcases represent an example, they can be utilized to demonstrate a new functionality of the application to customers. In this way, misunderstandings become preventable and users receive the chance to provide feedback and requests for modifications. Exploratory testing is simultaneously characterized as learning, test design and test execution (Abran et al, 2004). Here, a tester does not follow a specified test plan but instead dynamically designs, executes and modifies the checks. Therefore, in addition to revealing bugs, exploratory testing also helps generating a new set of automated checks and potentially offers input for additional requirements of the software (Humble and Farley, 2011). Usability testing is conducted to understand how easy it is for the user to operate and learn the application to accomplish his objectives (Abran et al, 2004). This includes user documentation, how effectively the system works in supporting user tasks, and its ability to recover from user errors. For example, surveys can serve for the quantification of this aspect. Another option is recording the user while working with the application to e. g. measure how much time is needed for certain processes or how well the interaction with the user interface takes place (e. g. clicking wrong buttons). Furthermore, some projects also employ beta testing programs to obtain more valuable feedback from a small and representative set of real customers (Abran et al, 2004).

The fourth quadrant consists of non-functional acceptance tests which refer to the review of the above-mentioned system parameters like performance (Abran et al, 2004), excluding functionality. These technical checks both are of both manual and automatic character (Humble and Farley, 2011). As they

often require dedicated test environments (Humble and Farley, 2011), non-functional acceptance tests are associated with high effort. Since, as initially indicated, the distinction to functional needs may not always seem clear-cut, alternative terms like "cross-functional requirements" or "system characteristics" are circulating in the literature (Humble and Farley, 2011). Regardless of the naming issue, it is essential to consider these tests from the beginning of the project. After all, detecting that the product has deficiencies (e. g. poor performance) shortly before the release date should be avoided under all circumstances.

3.4 The Role of SCATs in the Testing Pipeline

The aim of this subsection persists in characterizing SCATs, explaining their functional scope, pointing out deficits, and finally classifying them in the context of the Deployment Pipeline and the test quadrants. Thus, this section builds the foundation for comparing these tools according to the proposed method in the upcoming section. Due to the high time expenditure, manual code review reaches the limits of what is feasible (Boehm, 1984). For this reason, the static code analysis tools are supposed to support these reviews. The analysis of the code is performed without executing the actual program (the opposite case is called dynamic code analysis) (Novak et al, 2010). These validation tools either work directly on the source code (such as PMD) or on compiled byte code (e. g. FindBugs) (Marcilio et al, 2020). SCATs examine the program code with the goal of detecting possible defects that could lead to issues in a software system (Panichella et al, 2015). This involves the early discovery of potential errors, vulnerabilities, code smells or the assessment of compliance with coding standards and guidelines (Zampetti et al, 2017). A few more specific examples for possible findings are (Novak et al, 2010):

- syntactic errors
- unreachable source code
- undeclared variables
- non-initialized variables
- unused functions or procedures
- variables used before initialization
- non-use of values from functions
- wrong usage of pointers

SCATs use data and control flow analysis (Beller et al, 2016) to search for these particular patterns and rules in the source code (Novak et al, 2010). Overall, such tools have therefore emerged as an important pillar in modern software quality assurance procedures (Beller et al, 2016). Because of these so-called "warnings", development teams are empowered to identify and fix problems well in advance of the release versions. This reduces the effort for testing and the costs for system maintenance (Novak et al, 2010). However, it should not be neglected that SCATs are not suitable for universal application due to their dependence on the respective programming language(s) (Rahman et al, 2014).

The named type of errors indicate that SCATs are limited in their functionality. SCATs do not provide any verification of design or architectural flaws within the program to developers or testers. Therefore, these tools do not understand what the code is supposed to do (Novak et al, 2010). For this reason, they can just mitigate a handful of bugs so that other checks or manual testing remains essential. Even working with the SCATs does not present a trivial task. Apart from the potentially difficult deployment (Rahman et al, 2014), these validation tools do not completely substitute human code review. As a matter of fact, the verification of the properties of a software program

falls into the category of undecidable problems (Beller et al, 2016). This implies that SCATs cannot detect all shortcomings of the code and mark them as warnings (false negatives). On the other hand, some of these messages do not represent actual errors (false positives).

According to Zampetti et al (2017), the portion of false positives amounts to an astounding 95% among all issued warnings. Although developers generally rate SCATs beneficial, this number of false positives raises reservations (Johnson et al, 2013). Because of this, one is forced to manually check the findings in the source code for their justification. Consequently, SCATs should rather be regarded as a partial workload reduction since it is no longer necessary to manually process all lines of code. In this respect, false negatives can become dangerous and expensive, i. e. when potential defects are not classified as such (Rahman et al, 2014). Most software development projects merely use one SCAT, probably due to their requirement of cautious configuration (Beller et al, 2016) and non-trivial deployment (Rahman et al, 2014). In theory, such configurations can be utilized to limit the number of false positives. Some SCATs offer the capabilities to customize what kind of flaws is specifically searched for. The software may also allow the addition of new patterns that SCATs apply for defect detection (Johnson et al, 2013). However, in practice, the degree of "custom rules" seems kept within limits (Beller et al, 2016). Furthermore, modifications to the base configurations almost exclusively occur within the first week of setup. This fact suggests that more user-friendly possibilities for adaptations would be desirable for developers (Johnson et al, 2013).

Overall, the examined literature illustrates that SCATs should ideally be part of a well-defined and comprehensive test process. For example, Zampetti et al (2017) investigated SCATs in a CI pipeline, Beller et al (2016) recommend

the inclusion into a standard workflow (e. g. CI) to fully realize the potential of SCATs, and Novak et al (2010) emphasize the cooperation with other testing tools. In the context of the previously discussed Deployment Pipeline (Section 3.2), code analysis is placed as one of the initial steps performed by developers during the commit stage (Humble and Farley, 2011). According to the test quadrants from Section 3.3, checks via SCATs are assigned to the technology-facing tests. As per Brucker and Sodan (2014), SCATs can especially occupy an important role for detecting security vulnerabilities (one aspect of compliance with non-functional requirements). Accordingly, a SCAT does not represent a general-purpose tool in terms of software testing under any circumstances. However, the mentioned possibilities for application demonstrated that they become a pivotal part in a comprehensive CD testing process, since they are largely automated and can discover deficiencies (e. g. in security concerns) without program execution early in the development cycle (Brucker and Sodan, 2014). Nevertheless, the vast number of false positives, leading to manual effort for developers, together with the relatively small functional spectrum ensure that SCATs do not replace other tests in the Deployment Pipeline.

4 A Method for constructing a SCAT comparison catalog

In this section, the design of the artifact of our studies, the method for the construction of SCAT selection catalogs, is introduced. This corresponds to the "Design and Development" step of the DSR process (Peffers et al, 2007). Accordingly, RQ 2 is answered in the course of this. The method is divided into two subparts. First, guidelines for deriving comparison criteria for the tools are provided, using the findings related to RQ 1. The second parts deals with systematically and transparently determining the superior tool according to

the criteria. Therefore, requirements and application for the decision-making theory in this context are discussed.

4.1 Development of Evaluation Criteria

Software selection resembles a decision situation. Before discussing the potential ways to arrive at the superior tool, a comparison first of all requires the definition of criteria, allowing to contrast and evaluate the alternatives. Over time, various software quality models emerged for this purpose (e. g. ISO 25010). Overviews and comparisons of such models can be found in P. Miguel et al (2014) or in Gordieiev et al (2014), for instance. However, these models are designed to be somewhat universally applicable to all types of software, resulting in possibly disregarding important peculiarities of certain tool kinds that are essential for their comparison and selection. Accordingly, the plain usage of a software quality model to evaluate and compare SCATs is not recommended in this proposed method. Instead, to be able to utilize the quality models here, the previously discussed background of the check tools serves as the foundation to adjust the basic criteria to the specifics of SCATs.

This subsection summarizes the findings related to RQ 1 that thus should be taken into consideration when adapting the given criteria of a software quality model to construct a SCAT comparison catalog. When evaluating the functionality of a SCAT, special attention should be paid to supported programming languages (Rahman et al, 2014), the rate of false classifications (especially false positives) (Zampetti et al, 2017) and the type of detected code flaws. As SCATs possess limited capabilities in the range of bugs they can find, an application of these tools is often found in conjunction with other tests in the scope of a CI pipeline (Zampetti et al, 2017). Consequently, their interplay with other tools in a comprehensive test suite needs to be considered as

well. Accordingly, the deployment of a SCAT plays a factor in this case since it can prove to be difficult (Rahman et al, 2014). Additionally, SCATs might require meticulous configuration (Beller et al, 2016). Therefore, the feasibility and tool-side support for this task presents an important indicator when evaluating the usability of a SCAT.

SCATs rely on a set of rules to find defects in the code. Some tools allow to modify these patterns or even add new ones. Thus, the realization of this feature should be reviewed and taken into account, too. Another aspect related to user-friendliness is centered around the support in manual code review. As this type of verification cannot be completely omitted due to a usually high degree of false positives, a good SCAT should offer some functionalities (e. g. quick fixes for warnings) to increase the efficiency of this process. According to the SCAT literature, these are the aspects that should be considered in the original criteria of a software quality model to select a suitable SCAT from a given set of alternatives.

4.2 The Decision Making Procedure

After having a prescription for developing SCAT comparison criteria, a recipe for evaluating and then selection the superior tool of the respective project context remains. This process, the so-called decision, must be transparent as well as well-founded and not seem arbitrary. Accordingly, a SCAT evaluation catalog requires the application of a suitable decision-making method. A decision is composed of the three basic aspects of decision problem, alternatives, and criteria (Roy, 2016). Applying this concept to the subject of this paper, the decision problem corresponds to the search for the most suitable static code analysis tool. Here, the tools available in the selection represent the alternatives. A criterion is defined as any kind of information that allows

the evaluation and comparison of alternatives (Roy, 2016). The criteria aspect was highlighted in the previous subsection.

Before discussing the evaluation along the criteria, implementing the possibility to assign importance weights to the individual criteria presents a required precondition. Decision-makers across different projects might rate certain of the introduced SCAT features more significant than others. Consequently, this needs to be reflected through criteria weights that also influence the final decision. In the field of Multiple Criteria Decision Making (MCDM), the outcome is determined by the alternative value that is calculated using the weights and the evaluation in the criteria (Baker et al, 2002). Therefore, a decision-making procedure sets rules how to assign weights, rate the alternatives in the individual criteria and aggregate the individual scores and weights to a total value per alternative (Tzeng and Huang, 2011).

Mostly, the literature focuses on quantitative decision-making models. Examples are the Analytic Hierarchy Process (AHP), Simple Additive Weighting Method (SAW), Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) and VIKOR (VIseKriterijumska Optimizacija I Kompromisno Resenje (Serbian), translated = Multicriteria Optimization and Compromise Solution) (Tzeng and Huang, 2011). Basically, decision makers weight and evaluate the given criteria with numerical values in these methods. However, quantitative procedures are not suitable for the purpose of this work in their original state. In particular, the quantified estimation of weights and evaluation of the SCATs in the individual criteria appears problematic in the context of this project. The software quality models, which are adapted to compare SCATs on a given set of criteria, consist of several categories, which might also include additional subcategories. Although models for the numerical evaluation of certain software quality characteristics exist, crisp numerical

data are not available throughout all adapted criteria of the software quality models. This is due to the fact that for each of these criteria (as well as sub-criteria), the corresponding evaluation procedures would first have to be developed and selected before the actual implementation. Moreover, a necessary standardization or normalization of the measurement scales used in the different models would represent an additional requirement. Furthermore, the possibility of prioritizing criteria by decision makers on the basis of purely numerical weights raises further concerns. Finally, the choice of an appropriate range of values for the weight scale is not a trivial challenge, as it should not be set arbitrarily. Due to the aforementioned obstacles, the decision making for the SCAT selection method is recommended to be carried out with procedures using linguistic predicates. This refers to both the evaluation of the alternatives in each criterion and the weighting of these categories. Linguistic predicates are advantageous for decision makers in this regard, because the evaluation is more intuitive due to the easily comprehensible semantics. As human judgments like here are often vague and cannot necessarily express preferences with precise numerical values (Chen, 2000), linguistic assessment is preferred in this case.

5 Demonstration

The last section introduced the artifact of this work, a method for constructing a SCAT comparison catalog. Before continuing with the demonstration, a brief summary is given. First, a software quality model is selected and extended in alignment with the peculiarities of SCATs that were extracted from the literature (RQ 1). Then, a MCDM procedure that uses linguistic weighting and evaluation scores is selected. Afterwards, the MCDM method is applied

to calculate the evaluation scores for all candidate SCATs to determine the alternative to be selected for the specific use case.

In this section, the artifact is utilized to solve an artificial SCAT selection problem to demonstrate its applicability and effectiveness for solving the outlined problem in accordance with the DSR methodology (Peffers et al, 2007). Therefore, we initially present the used software quality model (ISO 25010) and how it is adapted to the SCAT context. Thus, our instance of a SCAT evaluation catalog can be constructed and is described next. Furthermore, the functionality of the corresponding MCDM method (Fuzzy TOPSIS) is introduced as a prerequisite for the demonstration. This is followed by the execution of the SCAT selection problem including the description of the test frame and tools, the allocation of the criteria weights, the discussion of the awarded evaluation scores, and the determination of the final results according to Fuzzy TOPSIS.

5.1 The ISO 25010 Standard Regarding Software Quality

The ISO 25010 is one of quite a few models for the properties of software quality (P. Miguel et al, 2014). As an advancement of the ISO/IEC 9126 standard, it contains the quality characteristics which should be regarded when evaluating a software product (International Organization for Standardization, 2011). It consists of eight key features which is shown in Table 1. Although the ISO 25010 is not untouched by criticism, the standard belongs to the top level in this field (Gordieiev et al, 2014). Therefore, we use ISO 25010 as the adapted software quality model for demonstrating the functionality of the artifact. However, the incorporation of similar quality models seems quite promising, too.

Table 1 ISO 25010 concerning software quality (International Organization for Standardization, 2011)

Software Product Quality

Functional Suitability	**Performance Efficiency**	**Compatibility**	**Usability**	**Reliability**	**Security**	**Maintainability**	**Portability**
Functional Completeness	Time Behaviour	Co-existence	Appropriateness	Maturity	Confidentiality	Modularity	Adaptability
Functional Correctness	Resource Utilization	Interoperability	Recognizability	Availability	Integrity	Reusability	Installability
Functional Appropriateness	Capacity		Learnability	Fault Tolerance	Non-repudiation	Analysability	Replaceability
		Operability	Recoverability	Authenticity		Modifiability	
		User Error Protection		Accountability		Testability	
		User Interface Aesthetics					
		Accessibility					

In the following, the eight characteristics of the standard will be briefly explained to provide the required understanding for the course of argumentation. When examining the descriptions for the individual categories, it is noticeable that these are kept rather generic. Functional Suitability indicates the degree to which a system fulfills the functions for meeting specified requirements for use under specific conditions (International Organization for Standardization, 2011). This metric comprises the sub characteristics functional completeness, correctness and appropriateness. The second pillar called "Performance Efficiency" covers the effectiveness of the software in relation to the amount of used resources. For example, this refers to response and processing times (time behavior) as well as the amount and type of resources used (resource utilization). Compatibility reflects the degree to which a product, system or component is able to exchange information with others (interoperability) and/or can perform its required functions when occupying the same hardware or software environment (co-existence) (International Organization for Standardization, 2011).

In the context of software, usability represents a frequently used term. Within the framework of ISO 25010, it is defined as the measurement of how well a system is operable by the user group to achieve clarified goals under consideration of effectiveness, efficiency and satisfaction (International Organization for Standardization, 2011). This includes aspects such as simplicity of operation and maintenance (operability), user error protection and also user interface aesthetics. Reliability indicates to which extent a software accomplishes its defined functions under certain conditions within a specified period of time. This category contains the important metrics of availability, fault tolerance and recoverability. Security evaluates how the system protects information or data so that people or systems are only allowed access to data

according to their appropriate level of authorization (International Organization for Standardization, 2011). The subcategories comprise common keywords like confidentiality, integrity, accountability or authenticity.

Maintainability describes the degree of effectiveness and efficiency to what extent a system can be modified with the intention of improvement, correction or adjustment to changes in environment or in requirements (International Organization for Standardization, 2011). Modularity, reusability, modifiability and testability present pivotal parameters at this stage. The last of the eight characteristics of ISO 25010 is the so-called portability. It determines how well a system, product or component can be transferred from a hardware, software or production environment to another. Here, adaptability, installability and replaceability are employed as key figures (International Organization for Standardization, 2011).

5.2 Development of the Criteria

Now, the individual categories of the ISO 25010 described in the previous section need to be adapted for the SCAT comparison catalog in connection with the compiled basics from chapter 3.4. Referring to the functional suitability of SCATs, the specific programming language within the project presents a focus, since the individual tools often have just a limited range of code languages that they support respectively (Rahman et al, 2014). Consequently, SCATs will be rated higher in this area if they cover a wider spectrum. The main aspect of the functional suitability consists of the scope of functions, i. e. in the type of errors that SCATs are able to detect (see e. g. Section 3.4) (Novak et al, 2010). At this point, differences between the tools also occur, which can be evaluated in the context of the particular project and test environment. A major weakness of SCATs lies in the considerable proportion of

false positives within the displayed "warnings" (Zampetti et al, 2017). Therefore, the lower the rate of misclassifications for a test tool amounts to, the better the evaluation of its functional suitability becomes.

Compatibility describes the ability to preserve the defined functions of a software when interacting with other systems. In terms of the SCATs, this refers primarily to the processes of Deployment Pipeline or Continuous Integration, where these tools are usually included (Zampetti et al, 2017). There, they form part of a comprehensive set of tests and must be able to work well together with the other checks in order to bring potential errors to light. Therefore, enabling a simplified integration into a test pipeline and cooperation with further validations on the SCAT side plays an in important role, provided that an employment in such a context is desired in the respective project (Beller et al, 2016).

In their studies, Johnson et al (2013) found that the usability of SCATs was sometimes criticized from the viewpoint of developers. The mentioned immense number of false positives ensures that a manual code review remains indispensable (Beller et al, 2016). Furthermore, research revealed a lack of informative messages that could help the developers in their assessment of the warnings (Johnson et al, 2013). So-called "quick fixes" which suggest an automatic correction for a detected error, are also discussed by Johnson et al. in this matter. SCATs showing themselves capable of improving these properties (e. g. in the GUI) are rated higher in this criterion.

With the aspect of maintainability, ISO 25010 basically means the simplicity in executing modifications with the goal of increasing software quality. For the static code analysis tools, this mainly relates to reduction of false positives and false negatives. In this perspective, SCATs may allow for the adaptations of rules for finding bugs and determining what kind of issues should not be

regarded any further. The fact that rather few of such customizations are performed (Beller et al, 2016), indicates the necessity for easier adjustments which would lead to a better evaluation in this category.

With regards to SCATs, portability with its subcategories simplicity of installation and replaceability particularly covers the aspect of deployment. According to Rahman et al., this can present a difficult obstacle (Rahman et al, 2014). The frequent use of these tools in test pipelines in connection with other test procedures increases the importance of the required installation effort. SCATs that support or facilitate their deployment by certain implemented features would earn a higher rating at this stage. Concerning the replaceability, it should be noted that the respective programming language can present a limitation considering which SCATs are available.

The remaining categories are less specific to SCATs, as they are linked to metrics for the operation of a software. Performance efficiency is utilized to evaluate, among other things, run times and resource consumption. With regard to SCATs, these checks normally occur fast because the code does not need to be executed for this purpose (Novak et al, 2010). Furthermore, SCATs are not liberated from the aforementioned reliability characteristics of software such as availability, recoverability and fault tolerance. Therefore, these are considered crucial parameters for the evaluation, too. Security remains as the last open part of ISO 25010. According to the presented investigations of Section 3.4, there are no particularities specific to SCATs in this aspect. However, this should not be viewed as a reduction of the importance of this quality criterion.

5.3 The SCAT Catalogue

After deriving the criteria, the comparison catalog for static code analysis tools can now be constructed and summarized. It is depicted in tabular form in the Tables 2 and 3. For this purpose, it consists of three columns. The first column contains the name of the respective main comparison criterion. This may have been renamed in contrast to ISO 25010 to better match the SCAT-specific content of a category. The same applies to the column of sub-criteria that exist for each category. Then, the description column textually summarizes which aspects per criterion are relevant for the final assessment on the basis of Section 3.4.

Table 2 SCAT comparison catalog, part 1

Criteria	Subcriteria	Description
Functionality		Considers the functional range of SCATs, which is characterized by the following categories. It is worth mentioning that the aspect of the required functionality mainly differs across projects (e. g. type of errors and programming language) and thus must be evaluated against the background of the respective test environment.
	Detected Errors	An integral share of the functionality is the amount and type of errors that can be detected in the code. At this stage, significant differences between the SCATs may occur.
	Supported Programming Languages	SCATs are primarily bound to the programming languages supported by the tool. Therefore, the respective of the tool presents a factor in the evaluation.
	Incorrect Classifications	Another criterion of this category is the number of misclassifications (false positives and false negatives), which sometimes turn undesirably high for these tools.
Performance Efficiency		This criterion includes the performance and resource utilization of the tool. Essential indicators for this can be found in the subcriteria.
	Response and Processing Times	SCATs should be able to perform fast checks because the validation is conducted without actually executing the code.
	Resource Utilization	This aspect comprises the amount and type of resources used.
	Capacity	Determines the maximum possible output of the respective tool.
Compatibility		Generally refers to the ability to maintain functionality when interacting with other systems. SCATs are often integrated into a comprehensive test framework such as Continuous Integration or the Deployment Pipeline in order to realize their maximum potential. This category evaluates this property.
	Integration Potential	The successful and functioning integration of these test tools and their cooperation with other tests (test pipeline) amounts to high importance in development projects. Consequently, if SCATs possess such capabilities that are conducive in this respect, they perform better in this category.
Usability		Describes in broad terms the quality of operability of a software. According to literature, SCATs score comparably poorly in this category.
	Support of Manual Code Review	The massive level of incorrect classifications still requires human code review. Therefore, this category evaluates among other things, to what extent the developers are assisted in manually checking the code as a result of the SCAT test. For examples, positives can contain the provision of informative messages or even quick fixes for warnings, which are intended to help the user to fix errors.
	Perception of Use	Covers whether the software is easy to control and comfortable to use.
	User Interface	Does the tool have an appealing and intuitive graphical interface?

Table 3 SCAT comparison catalog, part 2

Reliability		Indicates the extent to which the defined functions of the tool can be performed.
	Maturity	This aspect describes the fulfillment of stated requirements in normal operation.
	Availability	Measures whether the software is operational and accessible when needed.
	Fault Tolerance	Assesses the degree to which the operation works as intended even in the case of software or hardware deficiencies.
	Recoverability	Provides an evaluation about the degree to which affected data and the desired system state can be restored in case of an interruption or a crash.
Security		Measures the compliance with the common security aspects for software.
	Confidentiality	Denotes the restriction of data access to only authorized persons.
	Integrity	Indicates the extent wo which unauthorized changes are prevented.
	Non-repudiation	Expresses the provability of occurred actions.
	Authenticity	Comprises the confirmation of the identity of a person or resource.
	Accountability	Evaluates the traceability of actions of an entity to exactly this entity.
Modifiability		This criterion describes the extent to which changes can be introduced in a SCAT to improve its functional effectiveness. Usually, such customizations are made at a rather low frequency.
	Addition or Removal of Rules and Patterns	Refers to adjustments that can be utilized to add or exclude rules and patterns for defect detention in order to reduce false classifications (false positives and false negatives).
Deployment		This twofold category is intended to determine, among other things, how the SCAT under evaluation can be integrated into the respective development or test environment. Overall, the project-specific context should not be disregarded.
	Implementation	Evaluates the difficulty of installing such a tool, which according to the literature sometimes turns out problematic. It may be necessary to pay special attention in the context of holistic processes like the Deployment Pipeline or similar concepts.
	Replaceability	Indicates how well a SCAT can be replaced by another test tool to fulfill the same purpose.

Next, selecting a decision making procedure for weighting the criteria and evaluating the SCATs along them remains. Therefore, Fuzzy TOPSIS is presented in the upcoming subsection. For this instance of the comparison catalog, it should be noted that evaluation and weighting is only performed for the eight main categories. From the view of the decision makers, it is intended that the respective sub-criteria of course influence the score of the superordinated criterion. However, an individual rating and weighting of the sub-criteria is not envisaged. Overall, the balance between the feasibility of the decision process and its legitimacy through considering all relevant circumstances in detailed attention should always be preserved. In the opinion of the authors, the additional effort caused by also weighting and evaluating the subordinate categories would largely increase the complexity of the decision-making procedure and in no way justify the benefit gained. Nevertheless, for potential

future comparison catalogs constructed with this method, the possibility of sensibly implementing the individual consideration of sub-criteria in rating and prioritizing should not be ruled out.

5.4 Fuzzy TOPSIS as Decision-making Procedure

After combining the knowledge about SCATs with ISO 25010, the first version for a comparison catalog for SCATs with the proposed method was set up. Determining a suitable tool for a given project from a set of eligible SCATs requires a MCDM method. Section 4.2 outlined that conventional quantitative procedures are considered less adequate for the matter of this article. As a result, "Fuzzy TOPSIS", which represents an advancement of the classic TOPSIS, can be applied as a possible solution in this respect (Chen, 2000). In this regulation, the weighting of criteria and evaluation of the alternatives in these categories is performed based on linguistic terms. Then, these qualitative ratings are transformed into mathematical vectors to be able to calculate the winning alternative according to the original TOPSIS. In order to use this process in application within the SCAT comparison catalog, the individual steps of Fuzzy TOPSIS are explained in detail.

Table 4 Weighting matrix

Criteria	D_1	D_2	...	D_K
Functionality	w_1^1	w_1^2	...	w_1^K
Performance Efficiency	w_2^1	w_2^2	...	w_2^K
Compatibility	w_3^1	w_3^2	...	w_3^K
Usability	w_4^1	w_4^2	...	w_4^K
Reliability	w_5^1	w_5^2	...	w_5^K
Security	w_6^1	w_6^2	...	w_6^K
Modifiability	w_7^1	w_7^2	...	w_7^K
Deployment	w_8^1	w_8^2	...	w_8^K

The first step for the decision makers $(D_1, ..., D_K)$ is to weight the criteria $C_1, ..., C_n$ because not all categories for the evaluation always have the

same importance attached to them. For example, certain projects or companies would favor usability and compatibility versus performance efficiency. Thus, it must be possible to assign weights $w_1, ..., w_n$ to the individual criteria in the comparison of the tools which reflect the preferences of the decision makers. Table 4 includes the resulting generic weighting matrix. In order to avoid the disadvantages of a purely numerical weighting from the perspective of the deciders, Fuzzy TOPSIS offers seven variables for the assessment of the importance of a criterion, as Table 5 highlights (Chen, 2000).

Table 5 Linguistic variables for importance weight (Chen, 2000)

Name	Abbreviation	Vector
Very low	VL	(0, 0, 0.1)
Low	L	(0, 0.1, 0.3)
Medium low	ML	(0.1, 0.3, 0.5)
Medium	M	(0.3, 0.5, 0.7)
Medium high	MH	(0.5, 0.7, 0.9)
High	H	(0.7, 0.9, 1)
Very high	VH	(0.9, 1, 1)

With the help of these predicates, the priority of the individual criteria is determined. The actual rating of the alternatives $A_1, ..., A_m$ in the respective categories is performed similarly (x_{ij} denotes the rating of an alternative A_i in the criterion C_j). At this stage, the decision makers also have seven linguistic variables at their disposal to evaluate the performance of the alternatives with respect to the criteria. Table 6 illustrates them.

Table 6 Linguistic variables for rating (Chen, 2000)

Name	Abbreviation	Vector
Very poor	VP	(0, 0, 1)
Poor	P	(0, 1, 3)
Medium poor	MP	(1, 3, 5)
Fair	F	(3, 5, 7)
Medium good	MG	(5, 7, 9)
Good	G	(7, 9, 10)
Very good	VG	(9, 10, 10)

Table 7 shows the resulting rating matrix. Then, the assigned predicates for weighting and evaluating are substituted by their mathematical vector representation for the subsequent calculations.

Table 7 Rating matrix

Criteria	**Tools**	**Decision makers**		
		D_1	...	D_K
Functionality	A_1	x_{11}^1	...	x_{11}^K
	...	...	...	...
	A_m	x_{m1}^1	...	x_{m1}^K
Performance Efficiency	A_1	x_{12}^1	...	x_{12}^K
	...	...	...	...
	A_m	x_{m2}^1	...	x_{m2}^K
Compatibility	A_1	x_{13}^1	...	x_{13}^K
	...	...	...	...
	A_m	x_{m3}^1	...	x_{m3}^K
Usability	A_1	x_{14}^1	...	x_{14}^K
	...	...	...	...
	A_m	x_{m4}^1	...	x_{m4}^K
Reliability	A_1	x_{15}^1	...	x_{15}^K
	...	...	...	...
	A_m	x_{m5}^1	...	x_{m5}^K
Security	A_1	x_{16}^1	...	x_{16}^K
	...	...	...	...
	A_m	x_{m6}^1	...	x_{m6}^K
Modifiability	A_1	x_{17}^1	...	x_{17}^K
	...	...	...	...
	A_m	x_{m7}^1	...	x_{m7}^K
Deployment	A_1	x_{18}^1	...	x_{18}^K
	...	...	...	...
	A_m	x_{m8}^1	...	x_{m8}^K

In most cases, the set of decision makers (K) includes more than one person. In this instance, the averages of the three-dimensional grade vectors for the weights ($\tilde{w}_j = (w_{j1}, w_{j2}, w_{j3})$) and ratings ($\tilde{x}_{ij} = (a_{ij}, b_{ij}, c_{ij})$) are calculated (Chen, 2000):

$$\tilde{x}_{ij} = \frac{1}{K}\left[\tilde{x}_{ij}^{1}(+)\tilde{x}_{ij}^{2}(+)...(+)\tilde{x}_{ij}^{K}\right] \tag{1}$$

$$\tilde{w}_{j} = \frac{1}{K}\left[\tilde{w}_{j}^{1}(+)\tilde{w}_{j}^{2}(+)...(+)\tilde{w}_{j}^{K}\right] \tag{2}$$

The next step consists of normalizing the rating vectors of the individual criteria to create a comparable scale. Depending on whether the respective category reflects a benefit (B) or cost criterion (C), one of the Formulas 3 or 4 is utilized for the computation of the normalized vector $\tilde{r}_{ij}$ (Chen, 2000):

$$\tilde{r}_{ij} = \left(\frac{a_{ij}}{c_j^*}, \frac{b_{ij}}{c_j^*}, \frac{c_{ij}}{c_j^*}\right), c_j^* = \max_i c_{ij} \text{ if } j \in B; \tag{3}$$

$$\tilde{r}_{ij} = \left(\frac{a_j^-}{c_{ij}}, \frac{a_j^-}{b_{ij}}, \frac{a_j^-}{a_{ij}}\right), a_j^- = \min_i a_{ij} \text{ if } j \in C; \tag{4}$$

Now the weighted normalized decision matrix $\tilde{V}$ can be set up by elementwise multiplication of the rating vectors with the weight vectors according to Equation 5 (Chen, 2000):

$$\tilde{V} = [\tilde{v}_{ij}]_{m\times n}, \text{where } \tilde{v}_{ij} = \tilde{r}_{ij} \circ \tilde{w}_j \tag{5}$$

For determining the optimal solution, the distance to the fuzzy positive-ideal solution (FPIS) and the fuzzy negative-ideal solution (FNIS) must be calculated for each alternative in each criterion. FPIS and FNIS are displayed in 6 and 7 (Chen, 2000):

$$\text{FPIS: } A^* = (\tilde{v}_1^*, \tilde{v}_2^*, ..., \tilde{v}_n^*), \text{where } \tilde{v}_j^* = (1,1,1) \tag{6}$$

$$\text{FNIS: } A^- = \left(\tilde{v}_1^-, \tilde{v}_2^-, ..., \tilde{v}_n^-\right), \text{where } \tilde{v}_j^- = (0,0,0) \tag{7}$$

For the distance computation of each alternative to the optimum (d_i^*) and pessimum (d_i^-), the Equations 8 and 9 are prescribed (Chen, 2000). Formula 10 outlines how the distance between two triangular fuzzy numbers $\tilde{m}$ and $\tilde{n}$ will be derived.

$$d_i^* = \sum_{j=1}^{n} d\left(\tilde{v}_{ij}, \tilde{v}_j^*\right), \ i = 1,2,...,m \tag{8}$$

$$d_i^- = \sum_{j=1}^{n} d\left(\tilde{v}_{ij}, \tilde{v}_j^-\right), \ i = 1,2,...,m \tag{9}$$

$$d\left(\tilde{m}, \tilde{n}\right) = \sqrt{\frac{1}{3}\left[\left(m_1 - n_1\right)^2 + \left(m_2 - n_2\right)^2 + \left(m_3 - n_3\right)^2\right]} \tag{10}$$

Lastly, the dominant alternative results from the so-called closeness coefficient CC. On the basis of the distance calculations, it can be determined as follows (Chen, 2000):

$$CC_i = \frac{d_i^-}{d_i^* + d_i^-}, \; i = 1, 2, ..., m \tag{11}$$

The CC_i of an alternative A_i can range between 0 and 1. The closer an alternative is located to the optimum (FPIS) (or: the further away it is located from the FNIS), the closer the similarity coefficient approaches 1. Therefore, the alternative with the highest CC represents the winning solution according to the Fuzzy TOPSIS method (Chen, 2000). The application of this decision procedure for this version of the SCAT comparison catalog is conducted in the following artificial scenario for the demonstration within this chapter.

5.5 Description of the Test Frame

In the demonstration, we simulate a software selection problem with three static test tools to be used within a development project. For deciding between the available check tools, the constructed instance of the SCAT comparison catalog developed with the proposed method is applied here. In order to be able to draw appropriate evaluative conclusions regarding the functionality and other included parameters of the SCATs, code examples with errors or other inadequacies are required. For this reason, the code of the benchmark Bugs.jar is utilized (bugs-dot jar, 2018), a large-scale and manifold data set with 1158 bugs for repair purposes in Java programs (Saha et al, 2018). Details and background information concerning Bugs.jar are described Saha et al (2018). For conducting the validation, the test code was imported into the IDE Eclipse,

where the SCATs could then be installed via plug-ins. This approach allows for simple and fast implementation of the tools. The three selected SCATs include checkstyle (checkstyle, 2022), PMD (sourceforge.net, 2013) and FindBugs (FindBugs, 2015). As these three tools support the Java programming language, these SCATS are quite popular within their area of application. Moreover, they are free and can be used as plug-ins in Eclipse.

Checkstyle is intended to help developers to write Java code that complies to corresponding standards (checkstyle, 2022). The publishers advertise a high degree of configurability and support for almost all coding standards. Additionally, design problems with classes and methods as well as deficiencies in layout or formatting of the code should be detectable. On its website, PMD indicates exemplary bugs (empty try/catch/switch statements), "dead code" like unused variables or parameters and too complicated expressions (e. g. unnecessary if statements) as weaknesses, that PMD searches the code for (sourceforge.net, 2013). FindBugs, a SCAT with over one million downloads, divides its warnings into three categories: "correctness bug", "bad practice" and "dodgy" (FindBugs, 2015). Correctness bugs include obvious coding errors. In this case, bad practice refers to the violation of recommended and essential programming practices. "Dodgy" relates to the code style, noting the potentially confusing, unusual or error-prone writing of code. In this category, FindBugs acknowledges a higher rate of false positives (FindBugs, 2015).

After explaining the framework of the evaluation, the next step comprises the prioritization of the eight comparison criteria by the group of decision makers. For the validation at hand, this responsibility is limited to a single decision maker. Even with this minimal number and the resulting increased subjectivity in arriving at the decision, the applicability of the comparison catalog can still be demonstrated without significant restrictions.

Table 8 Importance weights for the criteria

Criterion	Weight	Vector
Functionality	H	(0.7, 0.9, 1)
Performance Efficiency	L	(0, 0.1, 0.3)
Compatibility	MH	(0.5, 0.7, 0.9)
Usability	VH	(0.9, 1, 1)
Reliability	H	(0.7, 0.9, 1)
Security	M	(0.3, 0.5, 0.7)
Modifiability	VH	(0.9, 1, 1)
Deployment	ML	(0.1, 0.3, 0.5)

5.6 Allocation of the Importance Weights

Seven linguistic predicates are available for weighting the eight criteria (see Table 5). Table 8 shows the assigned importance weights for the criteria of the SCAT comparison catalog. This distribution is briefly reasoned below for each category from the point of view of the decision maker.

The functionality often represents the central argument for the test tools. Therefore, this criterion is assigned a high weighting but the strongest predicate is avoided here. This can be justified by the fact that the functionality can at least be partially variable due to the possibility of including "custom rules". Thus, the category modifiability achieves the priority "very high", because the reduction of annoying false positives among the warnings may be enabled here. The same rating is awarded to usability which presents a decisive factor for the acceptance of any software within the end users. A similar significance is attributed to the criterion reliability. Even the most functionally convincing and user-friendly tool is unsuitable if its defined functions are not permanently available or if the software is error-prone and not executable.

Since SCATs are often used in combination with other tests in schemes such as CI, the compatibility should normally receive a high importance weight. However, in the present evaluation framework, the tools do cooperate with other review units, so that this priority has been slightly downgraded. Security also represents a critical software aspect. Due to the local usage, it is less

Table 9 Rating of the SCATs

Criterion	**Tools**		
	FindBugs	checkstyle	PMD
Functionality	MG	MP	F
Performance Efficiency	G	F	F
Compatibility	F	F	F
Usability	G	MP	G
Reliability	P	MP	MP
Security	F	F	F
Modifiability	G	MP	F
Deployment	G	G	G

relevant at this point. Given the fact that the SCATs were integrated only as plug-ins into the Eclipse development environemnt in the course of this demonstration, the deployment is regarded as less important, too. Furthermore, this stems from the fact of not needing to incorporate the tools into a holistic test process. As static code analysis tools work without executing the code and therefore mostly run fast anyway, the lowest priority was attached to performance efficiency.

5.7 Rating of the Tools

This subsection of the demonstration contains the assessments of the decision-maker in relation to the Open Source SCATs FindBugs, checkstyle and PMD considering the eight comparison criteria. Table 9 hosts the assigned evaluation predicates. In the following, the rationale for these ratings will be outlined with regard to the three tools in each of the categories. It should be noted that the score consists of both the experience of using the SCATs as plug-ins in the Bugs.jar project and external experience reports about the tools. This double-tracked approach aims to reduce the subjectivity of the evaluation.

First of all, emphasis should be again put on the fact that SCATs perform their checks on the not executed code. Thus, the kind of errors they can detect are limited (for examples see Section 3.4) (Novak et al, 2010). This also

affected the application of the tools within Bugs.jar. For the most part, the designated bugs were not detected with the SCATs, which indicated that these present more sophisticated errors. Once again, this circumstance impressively underlines the importance of a more complex test environment.

The criterion functionality shows clearly noticeable differences between the three tools. This can be immediately recognized when reviewing the project "oak-core" (sub-directory of the content management system Jackrabbit Oak) of the Bugs.jar with the mentioned SCATs in the default mode. While FindBugs provides only 13 warnings in the entire branch, PMD tags thousands of these so-called violations. Checkstyle "outperforms" both competitors and highlights potential errors in almost every line of code. Checkystyle controls the compliance with the Google Checks code guideline by default. When applying the Sun code standards, the tool does not quite reach this enormous amount of warnings. Nevertheless, this multitude of notes in the code in the default configuration of checkstyle may already have a deterrent effect on developers, especially considering that little to no adjustments are performed to SCATs with high probability (Beller et al, 2016). Overall, checkstyle focuses more on the visual quality of the code. The warnings mainly refer to incorrect indentation depths or too many characters per line. The usefulness of these type of findings may well be doubted. Moreover, checkstyle is not able to detect potential security vulnerabilities (Louridas, 2006). According to Louridas, FindBugs and PMD possess at least some capacities, as far as the discovery of security-related issues is concerned. Due to the low number of warnings even when using the finest granulation, FindBugs seemingly is conceived to achieve productivity and practical applicability. One reason for this potentially persists in the way FindBugs works. Since the software utilizes the compiled byte code instead of the source code as the basis for its checks, certain code parts that

would usually be worthy of being reported are eliminated. This might also reduce the rate of false positives which in the case of FindBugs, allegedly just amounts to about 50 % (Novak et al, 2010). However, with this architecture it must be considered that the opposite undesired event of increasing the false negatives possibly occurs. The third Java SCAT PMD also records a very large amount of violations. In contrast to checkstyle, due to the classification of the warnings in five categories with different respective colors, a better overview is provided. Taking the amount of markers in the test code, it becomes clear that the stylistic aspects of the code play an important role in PMD, too. Although FindBugs claims to also consider these faults, the implementation in the test case proves that FindBugs is significantly less strict in this area. Ultimately, the rating in this criterion depends on the preference resulting from the user's main focus in this regard. Since the decision maker aims to reduce the effort of manual code review as far as possible, FindBugs is awarded the dominant score in this category. With PMD and checkstyle, the immense number of warnings stands rather counterproductive to this goal.

In terms of performance efficiency, there were no serious anomalies probably due to the fact that the tools can perform quick checks as they do not need to execute code. However, FindBugs should be pointed out in this case. In contrast to the other SCATs, FindBugs works on the compiled bytecode and not directly on the source code. (Louridas, 2006). From this, it can be concluded that FindBugs has a performance advantage for larger programs with more code. Accordingly, FindBugs receives the best rating in this criterion.

Among other things, compatibility includes the extent to which certain capacities for integration in e. g. test pipelines for cooperation with other test units exist. This aspect experiences a rather low level of meaning in the presented test framework and thus will likely not become the decisive factor

for the final SCAT selection. All three test tools possess plug-ins for many of the most renowned development environments such as Eclipse or IntelliJ (mainly developed and maintained by third parties) for easier collaboration. Ultimately, based on the gained insights, no remarkable differences could be identified between FindBugs, checkstyle and PMD in this area.

On the other hand, the differences in the aspect of usability were in part significant. FindBugs represents the only one of the used SCATs that has a GUI (Satrijandi and Widyani, 2014). After the code check with FindBugs, detailed information considering the potential errors (e. g. category, line) can be displayed via xml report. Checkstyle features no such user-friendly facilities. The warnings are marked (at least with the plug-in version) exclusively at the respective line of code. Especially with the sheer number of code annotations, a greater level of support would have been desirable at this point. PMD surpasses checkstyle with regard to this criterion. A "violations overview" transparently presents the detected warnings filtered by project, package and file. Detailed information are also available here. In addition, the categorization of errors with the help of the colors facilitates human code verification. However, the mentioned classification of the errors does not always seem to be understandable at first glance.

Concerning reliability, the experience with the existing test frame revealed significant deficits among the SCATs. On a variety of the projects of Bugs.jar, the used tools simply appeared to not work at all. The possibility that just no warnings were found in these code sections can be ignored with great certainty due to the large number of potential errors in other branches. For FindBugs, it can especially be noted that despite the correct installation and operability in other projects, the tool did not even appear in the list under "Properties" in the unworkable cases, unlike checkstyle and PMD. An initial assumption

would be that the failure has project-specific backgrounds, since otherwise the SCATs functioned flawlessly. Anyway, due to these circumstances, the reviewed SCATs do not achieve good ratings in the current evaluation.

The security of the tools did not represent an overly important role in the context of this work. However, when used in real projects and companies, this should be given the utmost attention. Both the own usage and research on this factor revealed no apparent security-related issues in FindBugs, checkstyle and PMD. On a positive note, all three testing tools receive regular releases for improvement (Novak et al, 2010), which also helps to eliminate potential weaknesses.

In simple terms, the criterion of modifiability is intended to assess how well the functionality of the SCATs can be improved by the user. This includes the possibility to add one's own rules or patterns to strengthen the quality of the defect detection. According to Satrijandi and Widyani, this requirement is met by FindBugs, checkstyle and PMD (Satrijandi and Widyani, 2014). For the purpose of adaptability, an overview and explanation of all already existing rules of the checking tools seems helpful. All three show such documentation on their respective websites, although checkstyle's version receives some criticism (Satrijandi and Widyani, 2014). Since the SCATs are used as plug-ins in Eclipse, the support for the customizability directly in the IDE should be reviewed as well. Here, Find Bugs possesses various settings. In the "Reporter Configuration" one is able to select the granularity of the warnings (1-20), the confidence value for reporting, detected bug categories and the type of marking for the respective danger level of an error. On the other hand, the "Detector Configuration" contains all the rules applied in FindBugs. Besides the provision of information regarding the category and speed of a rule, these patterns can either be selected or deselected with the help of checkboxes. Checkstyle

Table 10 Decision matrix including fuzzy ratings and weights

Criterion	Weight	Tools		
		FindBugs	checkstyle	PMD
Functionality	(0.7, 0.9, 1)	(5, 7, 9)	(1, 3, 5)	(3, 5, 7)
Performance Efficiency	(0, 0.1, 0.3)	(7, 9, 10)	(3, 5, 7)	(3, 5, 7)
Compatibility	(0.5, 0.7, 0.9)	(3, 5, 7)	(3, 5, 7)	(3, 5, 7)
Usability	(0.9, 1, 1)	(7, 9, 10)	(1, 3, 5)	(7, 9, 10)
Reliability	(0.7, 0.9, 1)	(0, 1, 3)	(1, 3, 5)	(1, 3, 5)
Security	(0.3, 0.5, 0.7)	(3, 5, 7)	(3, 5, 7)	(3, 5, 7)
Modifiability	(0.9, 1, 1)	(7, 9, 10)	(1, 3, 5)	(3, 5, 7)
Deployment	(0.1, 0.3, 0.5)	(7, 9, 10)	(7, 9, 10)	(7, 9, 10)

does not offer much room for configuration in this form. In addition to the choice of the coding standard to be complied with, merely the potential to exclude certain file categories from controls is featured. New configurations can be incorporated via a file at "Local Check Configurations". Furthermore, in contrast to FindBugs and PMD, an overview of the ruleset is missing. PMD shows to be more similar to FindBugs in the area of modifiability, since predefined rules can be activated and deactivated by checkboxes. Unfortunately, the not always understandable categorization of the errors and the lack of information concerning the types of rules increase the difficulty of developing custom rules.

With regards to the deployment of the examined tools, it can be stated that the commissioning of the SCATs went without any problems. Presumably, this was due to the fact that an integration into an existing holistic test suite was not required. Additionally, using the SCATs as plug-ins enabled a simplified installation in Eclipse. In terms of replaceability, major weaknesses could not be identified, too. Both the parallel execution of the SCATs on the same project and switching between the active test tools on a branch was performed without encountering any difficulties or errors. All in all, this means that there were no noticeable differences in quality between FindBugs, checkstyle and PMD in this criterion.

5.8 Results

Once the alternatives (FindBugs, checkstyle and PMD) have been rated in the eight comparison criteria with the available linguistic grades, the most suitable solution for the decision problem can be determined. In the following, the individual calculation steps of the Fuzzy TOPSIS algorithm according to Chen (2000) are performed using the validation example. First, Table 10 illustrates the weights and ratings. In this case, the conversion of the linguistic predicates into the respective vector representation was already completed. After setting up the fuzzy decision matrix, the scores must be normalized. For this purpose, Formulas 3 and 4 are applied. Since all of the eight categories exclusively represent benefit criteria, merely Equation 3 is required in this instance. The exemplary calculation for the normalized vector of the tool "FindBugs" in the criterion "Functionality" is featured in Equation 12.

$$\begin{aligned} \tilde{r}_{11} &= \left(\frac{a_{11}}{c_1^*}, \frac{b_{11}}{c_1^*}, \frac{c_{11}}{c_1^*} \right), c_1^* = \max_1 c_{11} \\ \tilde{r}_{11} &= \left(\frac{5}{9}, \frac{7}{9}, \frac{9}{9} \right) \\ \tilde{r}_{11} &= (0.56,\ 0.78,\ 1) \end{aligned} \tag{12}$$

The determination of the remaining values of the fuzzy-normalized decision matrix depicted in Table 11 is performed in compliance with this pattern. Afterwards, the normalized scores are combined with the weights according to Formula 5. Equation 13 contains the exemplary computation for FindBugs in the category "Functionality".

Table 11 Fuzzy-normalized decision matrix

Criterion	Weight	Tools		
		FindBugs	checkstyle	PMD
Functionality	(0.7, 0.9, 1)	(0.56, 0.78, 1)	(0.11, 0.33, 0.56)	(0.33, 0.56, 0.78)
Performance Efficiency	(0, 0.1, 0.3)	(0.7, 0.9, 1)	(0.3, 0.5, 0.7)	(0.3, 0.5, 0.7)
Compatibility	(0.5, 0.7, 0.9)	(0.43, 0.71, 1)	(0.43, 0.71, 1)	(0.43, 0.71, 1)
Usability	(0.9, 1, 1)	(0.7, 0.9, 1)	(0.1, 0.3, 0.5)	(0.7, 0.9, 1)
Reliability	(0.7, 0.9, 1)	(0, 0.2, 0.6)	(0.2, 0.6, 1)	(0.2, 0.6, 1)
Security	(0.3, 0.5, 0.7)	(0.43, 0.71, 1)	(0.43, 0.71, 1)	(0.43, 0.71, 1)
Modifiability	(0.9, 1, 1)	(0.7, 0.9, 1)	(0.1, 0.3, 0.5)	(0.3, 0.5, 0.7)
Deployment	(0.1, 0.3, 0.5)	(0.7, 0.9, 1)	(0.7, 0.9, 1)	(0.7, 0.9, 1)

$$\tilde{V} = [\tilde{v}_{ij}]_{3\times 8}\,, \text{where } \tilde{v}_{ij} = \tilde{r}_{ij} \circ \tilde{w}_j \tag{13}$$

$$\tilde{v}_{11} = \tilde{r}_{11} \circ \tilde{w}_1$$

$$\tilde{v}_{11} = (0.56,\ 0.78,\ 1) \circ (0.7,\ 0.9,\ 1)$$

$$\tilde{v}_{11} = (0.39,\ 0.7,\ 1)$$

The other values for all remaining tools and criteria are calculated according to the same principle. The resulting "fuzzy-normalized and weighted" decision matrix is illustrated in Table 12.

Table 12 Fuzzy-normalized and weighted decision matrix

Criterion	Tools		
	FindBugs	checkstyle	PMD
Functionality	(0.39, 0.7, 1)	(0.08, 0.3, 0.56)	(0.23, 0.5, 0.78)
Performance Efficiency	(0, 0.09, 0.3)	(0, 0.05, 0.21)	(0, 0.05, 0.21)
Compatibility	(0.21, 0.5, 0.9)	(0.21, 0.5, 0.9)	(0.21, 0.5, 0.9)
Usability	(0.63, 0.9, 1)	(0.09, 0.3, 0.5)	(0.63, 0.9, 1)
Reliability	(0, 0.18, 0.6)	(0.14, 0.54, 1)	(0.14, 0.54, 1)
Security	(0.13, 0.36, 0.7)	(0.13, 0.36, 0.7)	(0.13, 0.36, 0.7)
Modifiability	(0.63, 0.9, 1)	(0.09, 0.3, 0.5)	(0.27, 0.5, 0.7)
Deployment	(0.07, 0.27, 0.5)	(0.07, 0.27, 0.5)	(0.07, 0.27, 0.5)

Then, for each SCAT, the distances to the FPIS (Formula 6) and the FNIS (Formula 7) must be determined. The necessary procedure is prescribed by

the Formulas 8-10. Equations 14 and 15 depict how the required distances for FindBugs are calculated in detail.

$$\textbf{FPIS: } d_1^* = \sum_{j=1}^{8} d\left(\tilde{v}_{1j}, \tilde{v}_j^*\right), \text{where } \tilde{v}_j^* = (1,1,1) \tag{14}$$

$$d\left(\tilde{v}_{11}, \tilde{v}_1^*\right) = \sqrt{\frac{1}{3}\left[(0.39-1)^2 + (0.7-1)^2 + (1-1)^2\right]} = 0.39$$

$$d\left(\tilde{v}_{12}, \tilde{v}_2^*\right) = ... = 0.88$$

...

$$d_1^* = 0.39 + 0.88 + 0.54 + 0.22 + 0.78 + 0.70 + 0.22 + 0.74 = \mathbf{4.47}$$

$$\textbf{FNIS: } d_1^- = \sum_{j=1}^{8} d\left(\tilde{v}_{1j}, \tilde{v}_j^-\right), \text{where } \tilde{v}_j^- = (0,0,0) \tag{15}$$

$$d\left(\tilde{v}_{11}, \tilde{v}_1^-\right) = \sqrt{\frac{1}{3}\left[(0.39-0)^2 + (0.7-0)^2 + (1-0)^2\right]} = 0.74$$

$$d\left(\tilde{v}_{12}, \tilde{v}_1^-\right) = ... = 0.18$$

...

$$d_1^- = 0.74 + 0.18 + 0.61 + 0.86 + 0.36 + 0.46 + 0.86 + 0.33 = \mathbf{4.40}$$

Then, Equations 16 and 17 show the computation of the FPIS and FNIS values for the second SCAT checkstyle.

$$\textbf{FPIS: } d_2^* = \sum_{j=1}^{8} d\left(\tilde{v}_{2j}, \tilde{v}_j^*\right), \text{where } \tilde{v}_j^* = (1,1,1) \tag{16}$$

$$d\left(\tilde{v}_{21}, \tilde{v}_1^*\right) = \sqrt{\frac{1}{3}\left[(0.08-1)^2 + (0.3-1)^2 + (0.56-1)^2\right]} = 0.72$$

$$d\left(\tilde{v}_{22}, \tilde{v}_2^*\right) = ... = 0.92$$

...

$$d_2^* = 0.72 + 0.92 + 0.54 + 0.72 + 0.56 + 0.70 + 0.72 + 0.74 = \mathbf{5.62}$$

$$\textbf{FNIS: } d_2^- = \sum_{j=1}^{8} d\left(\tilde{v}_{2j}, \tilde{v}_j^-\right), \text{where } \tilde{v}_j^- = (0,0,0) \tag{17}$$

$$d\left(\tilde{v}_{21}, \tilde{v}_1^-\right) = \sqrt{\frac{1}{3}\left[(0.08-0)^2 + (0.3-0)^2 + (0.56-0)^2\right]} = 0.37$$

$$d\left(\tilde{v}_{22}, \tilde{v}_1^-\right) = ... = 0.12$$

...

$$d_2^- = 0.37 + 0.12 + 0.61 + 0.34 + 0.66 + 0.46 + 0.34 + 0.33 = \mathbf{3.23}$$

Similarly to the previous formulas, Equations 18 and 19 present the calculation path for PMD, the third alternative.

$$\textbf{FPIS: } d_3^* = \sum_{j=1}^{8} d\left(\tilde{v}_{3j}, \tilde{v}_j^*\right), \text{where } \tilde{v}_j^* = (1,1,1) \tag{18}$$

$$d\left(\tilde{v}_{31}, \tilde{v}_1^*\right) = \sqrt{\frac{1}{3}\left[(0.23-1)^2 + (0.5-1)^2 + (0.78-1)^2\right]} = 0.54$$

$$d\left(\tilde{v}_{32}, \tilde{v}_2^*\right) = ... = 0.92$$

...

$$d_3^* = 0.54 + 0.92 + 0.54 + 0.22 + 0.56 + 0.70 + 0.54 + 0.74 = \mathbf{4.76}$$

$$\textbf{FNIS: } d_3^- = \sum_{j=1}^{8} d\left(\tilde{v}_{3j}, \tilde{v}_j^-\right), \text{where } \tilde{v}_j^- = (0,0,0) \tag{19}$$

$$d\left(\tilde{v}_{31}, \tilde{v}_1^-\right) = \sqrt{\frac{1}{3}\left[(0.23-0)^2 + (0.5-0)^2 + (0.78-0)^2\right]} = 0.55$$

$$d\left(\tilde{v}_{32}, \tilde{v}_1^-\right) = ... = 0.12$$

...

$$d_3^- = 0.55 + 0.12 + 0.61 + 0.86 + 0.66 + 0.46 + 0.52 + 0.33 = \mathbf{4.11}$$

Finally, Table 13 summarizes the resulting FPIS and FNIS values for the regarded use case.

Table 13 Distance calculation

	A^*	A^-
FindBugs	4.47	4.40
checkstyle	5.62	3.23
PMD	4.76	4.11

With these distances being derived, only the computation of the closeness coefficient for each tool remains. With the utilization of Formula 11, this leads to the following values: FindBugs achieves $CC_1 = 0.50$, the result for checkstyle is $CC_2 = 0.36$ and PMD reaches $CC_3 = 0.46$. The precise origin of this outcome can be comprehended by means of Equations 20-22.

$$\text{FindBugs: } CC_1 = \frac{d_1^-}{d_1^* + d_1^-} = \frac{4.40}{4.47 + 4.40} = 0.50 \tag{20}$$

$$\text{checkstyle: } CC_2 = \frac{d_2^-}{d_2^* + d_2^-} = \frac{3.23}{5.62 + 3.23} = 0.36 \tag{21}$$

$$\text{PMD: } CC_3 = \frac{d_3^-}{d_3^* + d_3^-} = \frac{4.11}{4.76 + 4.11} = 0.46 \tag{22}$$

Accordingly, as it attained the highest similarity coefficient of the three, FindBugs would be selected as the preferred alternative for the simulated SCAT decision problem in the demonstration of this comparison catalog instance. PMD comes in second place. On the other hand, checkstyle achieved the lowest score and thus was judged to be the least suitable static code analysis tool.

6 Evaluation

The motivation of the demonstration is to underline the efficacy of the artifact to solve the outlined problem regarding the SCAT selection process (Peffers et al, 2007). Therefore, the proposed method was applied to construct a SCAT comparison catalog, using the ISO 25010 standard for software quality as a basis and Fuzzy TOPSIS as the MCDM procedure. Then, a SCAT selection process was successfully simulated to contrast three SCATs according to the catalog. Accordingly, the applicability of the proposed SCAT comparison catalog was demonstrated by the means of an "Illustrative Scenario" (Peffers et al, 2012). As a consequence, this section comprises the evaluation according to the adopted DSR process of Peffers et al (2007). Thus, it is argumentatively outlined how the proposed artifact contributes to a solution of the initially stated problems and fulfills the objectives of a solution.

First of all, the introduced method aims to improve the SCAT selection process for tests in software engineering projects. One important factor represents the simplicity. If a respective method appears to be too complex and complicated, it is unlikely to be utilized in practice. Hence, we apply comparison criteria that are based on renown software quality models like ISO 25010. Additionally, the artifact requires the assignment of a MCDM procedure that

uses linguistic predicates (e. g. Fuzzy TOPSIS). Due to these easily comprehensible semantics, the evaluation is more intuitive for decision makers. This also contributes to transparency, since the usage of non-numerical weighting and rating scores is clearly understandable by each stakeholder. The application of a decision-making methodology in general presents a crucial factor for transparency because it legitimates the results and makes it reproducible. In contrast to an ad hoc and improvised SCAT selection, the construction of a comparison catalog according to the proposed method achieves a framework with clearly defined criteria that provides a traceable professional foundation for the SCAT selection process.

Another goal is the enablement of decision-makers to take project specifics into account. This is realized by allowing the assignment of weights to the derived criteria. These priorities are allocated with regards to the criterion's respective importance in the individual undertaking according to the opinion of a decision-maker. Consequently, the artifact of this work can be applied for any SCAT selection task in an arbitrary project. Therefore, the efficiency demand is covered, too, as the usage of a constructed SCAT catalog in a new project only requires the revision of the importance weights and evaluation scores of the SCAT alternatives. Furthermore, the design of the proposed method allows for adaption to other software decision situations with minimal effort. In comparison to the SCAT selection problem, merely the underlying theory for adjusting the respective categories of the chosen software quality model needs to be replaced with the context of the new application field.

With the introduction of the approach for enhancing the selection of a SCAT, it could be argued that the entire test process in software projects in general is also improved. However, SCATs only represent a small section of the comprehensive sequence that is required in software testing, especially when

considering automated approaches such as Continuous Integration and Deployment. Furthermore, the usage of an "Illustrative Scenario" (Peffers et al, 2012) in an artificial situation for the evaluation primarily focuses on showcasing the applicability of an DSR artifact. As a result, further experiments in real-world scenarios with comparisons to existing methodologies for addressing the matter of SCAT selection are necessary to definitively assess the value of this artifact.

7 Conclusion and Future Work

At the beginning of this paper, it was shown that the testing of software in the SDLC has become an immense task. To facilitate this process, concepts such as Continuous Integration or the Deployment Pipeline are available, which are intended to keep development cycles as small as possible and thus more manageable. Implementation also requires, among other things, code review by static code analysis tools (SCATs) at the beginning of the test pipeline. In order to support and simplify the selection of a suitable SCAT for an appropriate software project at this point, a method for deriving a SCAT comparison catalog based on software quality models and MCDM procedures was developed and validated using the DSR approach. In this last section, the results regarding the initially posed RQs are summarized and the limitations of our studies are outlined. In addition, an outlook on possible future work concludes this article.

7.1 Discussion of Research Questions

This section aggregates the results of the two research questions that were defined in the introduction. Thus, the contributions of this paper are summarized, including the key aspects of the artifact.

RQ1: What are the application fields of SCATs in (modern) software testing with regards to their capabilities, strengths and weaknesses?

Static code analysis tools check code without executing the actual program and are used to identify e. g. potential errors, vulnerabilities, and code smells to support manual code review (Novak et al, 2010). Accordingly these tools present an important pillar in modern software quality assurance (Beller et al, 2016). However, SCATs are limited in the types of errors that can be detected and display a high rate of false positives. Thus, they should be ideally part of a well-defined and comprehensive process like CI at an early stage (Zampetti et al, 2017)(Beller et al, 2016).

RQ2: Considering the findings from RQ 1, what should be the design of a systematic SCAT selection process, also taking into account the specific circumstances of the respective software development project?

The proposed method for solving the SCAT selection problem can be summarized as follows. First, a software quality model (e. g. ISO 25010) is chosen and adapted in alignment with the characteristics of SCATs that were extracted from the literature (RQ 1). Then, a MCDM procedure that uses linguistic weighting and evaluation scores is selected. Afterwards, the MCDM

method is applied to calculate the evaluation scores for all candidate SCATs to determine the alternative to be selected for the specific use case.

7.2 Limitations and Outlook

The artifact of this paper was demonstrated and evaluated in an artificial SCAT decision situation. Using ISO 25010 and Fuzzy TOPSIS, a SCAT comparison catalog was derived and utilized to contrast three SCATs (FindBugs, checkstyle, PMD). Then, it was argued how the method for the SCAT comparison catalog fulfilled the initially stated solution objectives. However, the artifact is subject to certain limitations, which will be mentioned in this concluding subsection.

First of all, the validation was only carried out with one decision maker, so that the full potential of the catalog was not shown. At some points in the evaluation, it became clear that the criteria could not always be clearly distinguished. For example, overlaps can be found in the categories of usability and adaptability. This fact can therefore be improved and may require a sharper definition. In addition, the determined values for the tools (similarity coefficients), which were relatively low at a maximum of 0.5, should not be regarded as representative because of the artificial situation. Furthermore, the use of the test tools as plug-ins plays a role in the origin of these rather lower scores. Moreover, the classification or description of the subcriteria, could also be enhanced. In order to maintain the balance between simplicity of implementation and additional benefit from individual weighting and scoring of the sub-points, the latter was omitted, as in the author's opinion this additional effort is not justified (see Section 4.2). Continued research could test this proposition by contrasting such individual evaluation in an appropriate decision procedure with the approach used here. Overall, in terms of the demonstration,

a content-related upgrade can be achieved through the inclusion of external expert opinions and a more in-depth examination of the tools. Additionally, further experiments in real-world scenarios with the proposed method need to be performed to draw conclusive comparisons to existing methodologies and to definitively assess the value of this artifact.

Due to the software quality model origin for the comparison criteria, the categories are predominantly high-level and include little technical depth. The pursuit of this approach was done to maintain a certain degree simplicity in the evaluation and selection, instead of increasingly complicating the whole process of decision making. The SCATs as the subject of the selection problem already indicate that this presents a strongly technical field, which in turn should arguably be reflected by the criteria. Another criticism is the lack of comparative categories for economic aspects such as acquisition and licensing costs, despite the rather abstract focus. Hence, using alternative sources for the development of the selection categories might be worth examining in future works. A benefit in the pursued approach with software quality models can be found in its portability to other software decision problems with minor modifications. Accordingly, further studies could research on the adaptation of this artifact to other areas.

Statements and Declarations

Author Contributions Statement

The corresponding author was responsible for writing the main manuscript text with the input of the other authors. All authors reviewed the manuscript (figures, text, and citations).

Funding Declaration

All authors declare that no funding was received for the creation and submission of this article.

Competing Interests

The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

References

Abran A, Moore JW, Bourque P, et al (2004) Guide to the Software Engineering Body of Knowledge. Swebok

Arusoaie A, Ciobaca S, Craciun V, et al (2017) A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC) pp 161–168

Baker D, Bridges D, Hunter R, et al (2002) Guidebook to decision-making methods

Beller M, Bholanath R, McIntosh S, et al (2016) Analyzing the state of static analysis: A large-scale evaluation in open source software. In: IEEE International Conference on Software Analysis E, Reengineering (eds) 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering. IEEE, Piscataway, NJ, pp 470–481, https://doi.org/10.1109/SANER.2016.105

Boehm BW (1984) Software engineering economics. IEEE transactions on Software Engineering pp 4–21

Bosch J (2014) Continuous software engineering: An introduction. In: Bosch J (ed) Continous software engineering. Springer, Cham, p 3–13, https://doi.org/10.1007/978-3-319-11283-1{_}1, URL https://doi.org/10.1007/978-3-319-11283-1_1

Brucker AD, Sodan U (2014) Deploying static application security testing on a large scale. Sicherheit 2014 – Sicherheit, Schutz und Zuverlässigkeit pp 91–101

Chatzieleftheriou G, Katsaros P (2011) Test-driving static analysis tools in search of c code vulnerabilities. IEEE 35th Annual Computer Software and Applications Conference Workshops pp 96–103

checkstyle (2022) About checkstyle. URL https://checkstyle.sourceforge.io/index.html

Chen CT (2000) Extensions of the topsis for group decision-making under fuzzy environment. Fuzzy Sets and Systems (Volume 114, Issue 1):1–9. URL https://doi.org/10.1016/S0165-0114(97)00377-1.

Chen L (2015) Continuous delivery: Huge benefits, but challenges too. IEEE Software (Vol. 32, No. 2):50–54. https://doi.org/10.1109/MS.2015.27

Crispin L, Gregory J (2009) Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley

FindBugs (2015) Findbugs™ - find bugs in java programs. URL http://findbugs.sourceforge.net/index.html

Fitzgerald B, Stol KJ (2017) Continuous software engineering: A roadmap and agenda. Journal of Systems and Software (Vol. 123):176–189. https://doi.org/10.1016/j.jss.2015.06.063, URL https://doi.org/10.1016/j.jss.2015.06.063.

Gordieiev O, Kharchenko V, Fominykh N, et al (2014) Evolution of software quality models in context of the standard iso 25010. Proceedings of the Ninth International Conference on Dependability and Complex Systems pp 223–232. URL https://link.springer.com/chapter/10.1007/978-3-319-07013-1_21

Hevner, March, Park, et al (2004) Design science in information systems research. MIS Quarterly (Vol. 28, No. 1):75–105. https://doi.org/10.2307/25148625, URL 10.2307/25148625

Humble J, Farley D (2011) Continuous delivery: Reliable software releases through build, test, and deployment automation. A Martin Fowler signature book, Addison-Wesley, Upper Saddle River, NJ

International Organization for Standardization (2011) Iso/iec 25010. URL https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&limitstart=0

bugs-dot jar (2018) Bugs.jar: A large-scale, diverse dataset of bugs for java program repair. URL https://github.com/bugs-dot-jar/bugs-dot-jar

Johnson B, Song Y, Murphy-Hill E (2013) Why don't software developers use static analysis tools to find bugs. 35th International Conference on Software Engineering (ICSE) pp 672–681. URL 10.1109/ICSE.2013.6606613

Kaur A, Nayyar R (2020) A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. Procedia Computer

Science (Vol. 171):2023–2029. https://doi.org/10.1016/j.procs.2020.04.217, URL 10.1016/j.procs.2020.04.217

Leppänen M, Mäkinen S, Pagels M, et al (2015) The highways and country roads to continuous deployment. IEEE Software (Volume: 32, Issue 2) pp 64–72

Louridas P (2006) Static code analysis. IEEE Software (Vol. 23, No. 4):58–61. URL 10.1109/MS.2006.114

Lunn K (2003) Software development life cycle. In: Lunn K (ed) Software Development with UML. Macmillan Education UK, London, p 53–68, https://doi.org/10.1007/978-0-230-80419-7{_}5

Marcilio D, Furia CA, Bonifácio R, et al (2020) Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings. Journal of Systems and Software (Vol. 168):110,671. https://doi.org/10.1016/j.jss.2020.110671, URL 10.1016/j.jss.2020.110671

Marick B (1998) When should a test be automated? Proceedings of The 11th International Software/Internet Quality Week pp 1–20

Meyer M (2014) Continuous integration and its tools. IEEE Software (Vo. 31, No. 3):14–16. https://doi.org/10.1109/MS.2014.58

Novak J, Krajnc A, Zontar R (2010) Taxonomy of static code analysis tools. The 33rd International Convention MIPRO pp 418–422

P. Miguel J, Mauricio D, Rodríguez G (2014) A review of software quality models for the evaluation of software products. International Journal of Software Engineering & Applications (Vol. 5, No. 6):31–53. https://doi.org/10.

5121/ijsea.2014.5603, URL 10.5121/ijsea.2014.5603

Panichella S, Arnaoudova V, Di Penta M, et al (2015) Would static analysis tools help developers with code reviews? IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER) pp 161–170. https://doi.org/10.1109/SANER.2015.7081826, URL 10.1109/SANER.2015.7081826

Peffers K, Tuunanen T, Rothenberger MA, et al (2007) A design science research methodology for information systems research. Journal of Management Information Systems (Vol. 24, No. 3):45–77. https://doi.org/10.2753/MIS0742-1222240302, URL 10.2753/MIS0742-1222240302

Peffers K, Rothenberger M, Tuunanen T, et al (2012) Design science research evaluation. Design Science Research in Information Systems Advances in Theory and Practice pp 398–410

Rahman F, Khatri S, Barr ET, et al (2014) Comparing static bug finders and statistical prediction. Proceedings of the 36th International Conference on Software Engineering pp 424–434. URL https://doi.org/10.1145/2568225.2568269

Roy B (2016) Paradigms and challenges. In: Greco S, Ehrgott M, Figueira JR (eds) Multiple criteria decision analysis. International series in operations research & management science, Springer, New York and Heidelberg and Dordrecht and London, p 19–39, https://doi.org/10.1007/978-1-4939-3094-4{_}2

Saha RK, Lyu Y, Lam W, et al (2018) Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: Zaidman A, Kamei Y, Hill E (eds)

2018 ACM/IEEE 15th International Conference on Mining Software Repositories. The Association for Computing Machinery and IEEE Computer Society Conference Publishing Services (CPS), New York, New York and Los Alamitos, California, pp 10–13, https://doi.org/10.1145/3196398.3196473

Satrijandi N, Widyani Y (2014) Efficiency measurement of java android code. International Conference on Data and Software Engineering pp 1–6

Shahin M, Babar MA, Zhu L (2017) Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access (Vol. 5):3909–3943. URL 10.1109/ACCESS.2017.2685629

sourceforge.net (2013) Welcome to pmd. URL https://pmd.sourceforge.io/pmd-5.0.4/

Staegemann D, Volk M, Lautenschläger E, et al (2021) Applying test driven development in the big data domain – lessons from the literature. In: 2021 International Conference on Information Technology (ICIT), pp 511–516, https://doi.org/10.1109/ICIT52682.2021.9491728

Tzeng Gh, Huang Jj (2011) Multiple attribute decision making: Methods and appliations. A Chapman &amp Hall book, CRC Press, Boca Raton, Fla.

vom Brocke J, Maedche A (2019) The dsr grid: six core dimensions for effectively planning and communicating design science research projects. Electronic Markets (Vol. 29, No. 3):379–385. https://doi.org/10.1007/s12525-019-00358-7, URL 10.1007/s12525-019-00358-7

Weber I, Nepal S, Zhu L (2016) Developing dependable and secure cloud applications. IEEE Internet Computing (Volume 20, Issue 3) pp 74–79

Zampetti F, Scalabrino S, Oliveto R, et al (2017) How open source projects use static code analysis tools in continuous integration pipelines. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) pp 334–344. URL 10.1109/MSR.2017.2