International Journal of
Information Systems and Project Management
ISSN (print):2182-7796, ISSN (online):2182-7788, ISSN (cd-rom):2182-780X
Available online at ijispm.sciencesphere.org

Tools for monitoring software quality in information systems development and maintenance: five key challenges and a design proposal

Rolf-Helge Pfeiffer
IT University of Copenhagen, Research Center for Government IT
Rued Langgaards Vej 7, 2300 Copenhagen
Denmark
ropf@itu.dk

Jon Aaen
IT University of Copenhagen, Danish Institute for IT Program Management
Rued Langgaards Vej 7, 2300 Copenhagen
Denmark
jonaa@itu.dk

Abstract:
As software grows in size and complexity, organizations increasingly apply tools to automatically assess software quality of information systems during development and maintenance. Software quality assessment tools (SWQAT) promise fast and actionable insights into the technical state of software through various quality characteristics, such as maintainability, reliability, or security. These tools have been used to support a wide variety of Information Technology (IT) project management decisions related to system development, contract negotiations, project terminations, and even settling legal disputes between suppliers and clients. However, despite their rising importance, questions regarding how they function and how reliable they are to support decision-making have so far escaped scholarly attention. This paper conducts an evaluation of widely used SWQATs and analyzes how they rate the quality of software systems of varying sizes, functionalities, and programming languages. Our results reveal five key challenges for using SWQATs in IT projects. To address these challenges, we propose a design for tailorable SWQATs that allows for more conscious and prudent software quality assessments that better reflect the socio-technical aspect of software systems and the context-specific nature of software quality.

Keywords:
software quality assessment tools; information systems management tools; software quality; project control and monitoring.

DOI: 10.12821/ijispm120102

Manuscript received: 22 September 2023
Manuscript accepted: 5 March 2024

1. Introduction

World-class companies use software quality assessment tools (SWQAT) for information systems development and maintenance:

> *"We're using CAST [a software quality assessment tool] to measure the quality both as we're rolling things out in terms of our new code as well as looking at the legacy code that has come from the different partners to develop our core solutions over the last five or six years. This enables us to focus our mediation efforts on determining where we need to go back and deal with our quality issues. For us, an IT outage or a defect for something that's going into production can mean the trucks can't roll from the distribution center (...) and just a single day outage means multiple millions of dollars of lost revenue."*
> - Software Quality Assurance and Test Director, Coca-Cola Bottling and Investment Group [1]

Ensuring software quality is central to developing and maintaining Information Systems (IS) [2]. However, as the number and complexity of systems increase, it becomes progressively more demanding to continuously assess and monitor software quality through manual inspections. Consequently, organizations increasingly rely on tools to automatically rate the technical state of their software systems according to various quality characteristics such as maintainability, reliability, or security [3,4,5]. Contemporary software quality assessment tools (SWQAT) perform static analysis of a software system's source code to detect bugs, track technical debt, and ensure that coding and compliance standards are met [5]. Thereby, SWQATs can help developers detect potential weaknesses in the code that might lead to vulnerabilities or failure [6] while also providing managers with an overall understanding of a system's technical state [7].

SWQATs are used for a wide variety of purposes, including developing large-scale industrial software systems [8], as a basis for consultancy reports [3], contract negotiations [9,10], project terminations [11,12], and even settling legal disputes between suppliers and clients [13]. However, despite their growing importance for developing and maintaining information systems, IS research has paid little attention to the reliability of SWQATs, how they function, what they measure, and how suitable they are to support decision-making [6]. In this paper, we investigate these problems guided by the following research questions:

1. *What are the key challenges of using software quality assessment tools (SWQATs) to support decision-making in information systems development and maintenance?*
2. *How can SWQATs be designed to address these challenges?*

To address these research questions, we evaluate six prominent and widely used SWQATs. First, we map the fundamental features of each tool to analyze how they assess software quality and which metrics they use. Next, we analyze to which degree SWQAT ratings are consistent and comparable by testing the tools on a variety of software systems of different sizes, functionalities, and programming languages. Our results highlight five key challenges that cast doubt on the reliability of using contemporary SWQATs for information system development and maintenance. To overcome these challenges, we propose a design for tailorable SWQATs that allows for more conscious and prudent software quality assessments that better reflect the socio-technical aspect of software systems and the context-specific nature of software quality.

In the remainder of the paper, we first present extant literature on SWQATs and provide empirical examples of how SWQATs are used in information system development and maintenance. Next, we describe our research design and method before presenting our findings and design proposal. We conclude by discussing implications for research and practice. Our work expands current research by adding to the scarce literature on SWQATs in Information Technology (IT) project management.

2. Background: Software quality assessment tools in IS development and maintenance

Software quality has been a long-time concern in IT management, and organizations invest considerable resources in ensuring software quality during software development, maintenance, and evolution [5, 14, 15]. Poor software quality can have devastating consequences for multiple stakeholders, including user dissatisfaction, economic losses, and long-term repercussions on a company's reputation [16, 17, 18]. Conversely, high software quality translates into fewer defects, less technical debt, and higher end-user satisfaction with a software system [4]. As such, it is critical for IT managers to ensure appropriate and ongoing assessment of the quality of their software systems throughout their lifecycle [2, 3, 5, 19]. Yet, software quality is an abstract and multidimensional concept that has been subject to much debate regarding how to define and measure software quality [6, 20, 21].

Some scholars primarily conceptualize software quality through external metrics such as user satisfaction [22, 23, 24], while other scholars have focused on internal characteristics of the source code in terms of structure, complexity, etc. [3, 14, 25]. Industrial standards for software quality, like the ISO/IEC 25000, typically incorporate a combination of external and internal quality characteristics related to functionality, reliability, usability, efficiency, maintainability, etc. [26]. However, continuous software quality assessment based on such standards can be a cumbersome and costly task for organizations with large software portfolios. While project documentation can be helpful, it alone is rarely sufficient for developer teams to maintain a software system [27]. As software systems grow in number and complexity, manual software quality assessment becomes progressively more demanding. Moreover, it is notoriously difficult to balance speed and quality in software development [15]. For instance, if software developer teams feel obliged to deliver demonstrable results in a short timeframe, they can be tempted to employ less rigorous software quality assessments, resulting in a pileup of quality-related challenges later on [28]. Therefore, organizations increasingly turn to SWQATs to provide efficient, effective, and timely quality assessments on a large scale [4, 5, 29].

2.1 What are software quality assessment tools (SWQATs)?

There exists a plethora of commercial and non-commercial SWQATs [6]. Most SWQATs are marketed to organizations with a promise to deliver fast and actionable insights during software development and maintenance [7, 30]. SWQATs are based on a set of characteristics and metrics that form a predefined reference system for their assessments [3]. SWQATs use this reference system to examine source code, detect defects, and measure metrics like lines of code (LOC) or amount of redundancy ("code clones"). Typically, SWQATs then aggregate their results into higher-level quality attributes like maintainability, reliability, complexity, quality in general, etc., and summarize the results in "quality reports" [5, 6].

2.2 The use of SWQATs in IS development and maintenance

SWQATs play an essential role in the IT consultancy industry [3]. For instance, the Software Improvement Group (SIG) relies on SWQATs when advising industrial clients and governmental agencies by turning SWQAT ratings into *"actionable advice using a combination of our technology, scientific methodologies and software engineering expertise to measure, monitor and analyze the source code and architecture of your applications"* [31].

The Chinese company Alibaba – one of the world's largest internet companies and retailers – uses SWQATs for continuous and automatic quality assessment of their software systems since manually assessing their multiple billion lines of code is no longer feasible [8]. SWQATs are applied to enhance efficiency and address the escalating demand for software quality assessments within the company. Now, IT managers at Alibaba use SWQATs to attain *"evidence from an independent assessor and having an external party to announce unpleasant truths (...) [and] to help developers gain reputation from high-quality code written by them"* [8, p. 148].

SWQATs are also used in the public sector. For instance, The Danish Agency for Digitization prescribes a model for portfolio management of central government IT systems [32], which requires all public agencies in Denmark to map the technical state of central government IT systems every three years. This has led the Danish tax agency, the court

administration, the municipal IT project organization KOMBIT, and many others to rely on commercially available SWQATs to assess the quality of their software systems [9, 11, 12].

However, while extant literature predominantly highlights the potential of automatically assessing software quality and a large scale [3, 5], it is not evident to which degree the application of SWQATs actually translates into better software quality [33]. Furthermore, research has paid limited attention to the quality characteristics and metrics that form the reference system for contemporary SWQATs. Consequently, little is known about how SWQATs function, what they measure, or how reliable they are as inputs for decision-making.

3. Research method

We adopt a design science approach [34] to guide our research on key challenges of using SWQATs to support decision-making in information systems development and maintenance, and how such tools can be designed to address these challenges. We conduct a technical evaluation of six prominent SWQATs based on Venable et al.'s Framework for Evaluation in Design Science [35]. The evaluation assesses how the SWQATs measure software quality and compares their ratings across different software systems. The results highlight key challenges for using SWQATs and lay the groundwork for designing a solution that addresses these challenges.

3.1 Data collection

We included six popular and widely used SWQATs in our evaluation. SWQAT 1 (Better Code Hub) is developed by a leading IT consultancy firm, the Software Improvement Group, whose clients include DHL, Philips, and KLM. SIG's use of SWQATs has previously been studied in both public and private use contexts [3]. The specific product has since been discontinued after our data collection in favor of Sigrid, a similar SWQAT, based on similar quality model, and produced by the same company [36]. SWQAT 2 (CAST Highlight), is another frequently used and discussed tool [7] developed by the global IT consultancy firm CAST, whose clients include FedEx, IBM, and Coca-Cola [37]. SWQAT 3-5 (Codacy, Codebeat, and Code Climate) are popular SWQATs that are integrated tightly to code hosting platforms like GitHub, to support developers and organizations to continuously assess quality of their products under development. Finally, we included the open-source tool SWQAT 6 (SonarQube) as this tool has been reported to be the most popular tool among practitioners and the most discussed tool in online software developer communities [7]. Taken together, the six tools included for analysis display a diverse array of leading examples of contemporary SWQATs.

For each SWQAT, we collected documentary data sources from official websites, books, and in-tool guidelines. Table 1 provides an overview of the SWQATs in this evaluation and information about their respective vendors.

Table 1: Overview of SWQATs in this evaluation and information about their vendors.

SWQATs	Description	Company	License	Documents
SWQAT 1 (Better Code Hub)	Cloud-based source code analysis service to support building maintainable software.	Software Improvement Group B.V. HQ: Amsterdam	Commercial (Paid individual plan)	Academic studies [3, 36] In-tool + online documentation [38]
SWQAT 2 (CAST Hightlight)	Automatic source code analysis tool that assesses technical debt, complexity, and application size, to improve software quality.	CAST SA HQ: Paris/New York	Commercial (Full access granted)	Online documentation [39]
SWQAT 3 (Codacy)	Cloud-based source code analysis service to monitor code quality.	Codacy HQ: Lisbon	Commercial (Free plan)	Online documentation [40]

SWQATs	Description	Company	License	Documents
SWQAT 4 (Codebeat)	Cloud-based source code analysis service to monitor code quality.	Code quest sp. z o.o. HQ: Warsaw	Commercial (Free plan)	Online documentation [41]
SWQAT 5 (Code Climate Quality)	Cloud-based source code analysis service to assess code maintainability.	Code Climate, Inc HQ: New York	Commercial (Free plan)	Online documentation [42]
SWQAT 6 (SonarQube)	On premise source code analysis service to assess code quality.	SonarSource S.A HQ: Geneva	Open source (Free access)	Academic studies [25, 30] Online documentation [43]

To analyze and compare the SWQATs, we let them assess the quality of six software systems of varying size, functionality, and primary programming language. These software systems are open-source and hosted on GitHub. We gathered the software repositories (source code, documentation, etc.) to create identical versions of the software systems as input for the SWQAT and thereby ensure comparability of results. For each software system, we collected the assessment results from the SWQATs' user interfaces with screenshots. Our reproduction kit [44] includes technical details, documentation screenshots, and links. Table 2 lists the software systems whose quality is assessed by the SWQATs.

Table 2: Overview of the software systems used as input for analyzing and comparing the SWQATs.

Software systems	Description	Programming language
Software system 1: Apache Ignite [45]	High-performance distributed in-memory database management system.	Java
Software system 2: Apache Commons VFS [46]	Virtual File System library.	Java
Software system 3: Apache Airflow [47]	Workflow management platform.	Python
Software system 4: Apache Superset [48]	Business intelligence web application.	Python
Software system 5: Apache CordovaJS [49]	Unified JavaScript layer for Apache Cordova projects.	JavaScript
Software system 6: Apache Echarts [50]	Charting and visualization library.	JavaScript

3.2 Data analysis

We evaluated the SWQATs in four steps. First, we examined documentary data to identify a) how each SWQAT measures software quality in terms of metrics they apply and their unit of analysis, and b) how final assessment results are generated with the underlying software quality model. Second, we let each SWQAT assess the quality of each of the six software systems. Third, we compared the assessment results across SWQATs. To compare SWQATs, we collected which quality characteristics are reported in the final quality assessment reports. We grouped quality characteristics across tools (based on names and descriptions in supplementary material) into overarching categories: Overall product quality, Maintainability, Security, Size, Complexity, and Duplication. For example, SWQAT 2 denotes the overall quality of a system as *"Health"*, SWQAT 3 denotes it as *"software quality grade"*, and SWQAT 4 as *"global project score"*. Since each of these describes the overall quality, we categorized them accordingly and compared the SWQATs along respective quality characteristics, see Table 4. Finally, we condensed the results into the five key challenges that need to be taken into account when using SWQAT in IT management. To address these challenges, we propose a design for tailorable SWQATs that allows for a more conscious, prudent, and context-specific application of software quality assessments.

4. Results

We present the results from our evaluation as five key challenges for using SWQATs in IT management.

4.1 Challenge 1: SWQATs measure source code quality — not software quality

As listed in Table 3, all six SWQATs solely focus on source code as the unit of analysis. This means that they are confined to only assess internal quality characteristics such as the structure and complexity of the source code. Consequently, their assessment completely leaves out external characteristics related to how the software works in its environments, the degree to which a software system meets requirements, data quality, intended usage, etc. Furthermore, the SWQATs neglect configuration files, even though such files can be considered as declarative code that alters systems and their behavior [51]. Likewise, the quality of natural language documents, such as requirements specifications or documentation, is also not assessed by any of the SWQATs. The only exception is SWQAT 6, which to a limited extent measures the frequency of comments in source code. Also, as can be noted in the table, there are substantial differences in the supported programming languages covered in each tool, ranging from 10 to 31 languages.

Table 3: SWQATs' unit of analysis and supported programming languages.

SWQATs	Unit of analysis	Supported programming languages
SWQAT 1	Static analysis of source code	17 languages: C#, C++, Go, Groovy, Java, JavaScript, Objective/C, Perl, PHP, Python, Ruby, Scala, Shell Script, Solidity, Swift, TypeScript, Kotlin
SWQAT 2	Static analysis of source code	27 languages: ABAP, ASP, Bash, C, C#, C++, Cobol, Csh, CSS, EGL, Flex, Fortran, HTML, Java, JavaScript, JScript, Ksh, PHP, PL/1, Python, SQL, TypeScript, VB.NET, VBScript, Visual Basic, XHTML
SWQAT 3	Static analysis of source code	31 languages: Apex, C, C#, C++, CoffeeScript, Crystal, CSS, Dockerfile, Elixir, Go, Java, JavaScript, JSON, JSP, Kotlin, LESS, Markdown, PHP, PLSQL, Powershell, Python, Ruby, SASS, Scala, Shell Script, Swift, TypeScript, Velocity, Visual Basic, VisualForce, XML
SWQAT 4	Static analysis of source code	10 languages: Elixir, Go,Java,Javascript,Kotlin, Objective-C, Python, Ruby, Swift, TypeScript
SWQAT 5	Static analysis of source code	11 languages: C#, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Scala, Swift, TypeScript
SWQAT 6	Static analysis of source code	27 languages: ABAP, Apex, C#, C, C++, COBOL, CSS, Flex, Go, Java, JavaScript, Kotlin, Objective-C, PHP, PLI, PLSQL, Python, RPG, Ruby, Scala, Swift, TypeScript, TSQL, VB.NET, VB6, HTML, XML

Usually, the SWQATs function by collecting all files that contain source code in one of the supported languages and parsing their contents into an internal representation, often into abstract syntax trees. From here, the SWQATs perform static analysis, which means assessing the code as written, without actually running it. The static analysis is based on a set of predefined rules to search for patterns in source code that are bad coding practices, error-prone, etc. An example of such a rule is SWQAT 1's *"Write Short Units of Code"*, which identifies functions and methods in source code that are longer than 15 lines of code (LOC). If too many units exceed this number, the rule is violated and reported accordingly. Another example is SWQAT 2's Java code rule *"Switch cases without ending breaks are hard to understand"* [52] which flags all switch statements where case clauses are not finalized with a respective statement. We analyze more examples of such rules and how the SWQATs implement them differently into metrics in section 4.3.

In sum, a key limitation of SWQATs is their restricted focus on static analysis of source code quality rather than providing a comprehensive assessment of the overall software quality of systems in use. While code quality is an essential aspect of software quality, it represents just one dimension of a broader concept. Being unaware of the system's specific context and requirements, SWQAT may flag code as problematic based on general rules or patterns without considering the unique needs of the application. Consequently, SWQATs operate on a quite narrow and fine-granular basis. While this approach can provide some insights into adherence to coding standards, they are limited in their ability to uncover runtime- and context-specific issues.

4.2 Challenge 2: SWQATs are based on low-level static analysis of single source files and disregard interaction between higher-level components

SWQATs base their quality assessments on low-level static analysis of single files authored in single programming languages. Thereby, they disregard that software systems are often constructed out of many interacting components written in various programming languages, which at runtime, interact via a plethora of protocols. Components are typically considered to be reusable and separately deployable units of software [53].

However, software is usually made up of multiple components composed of multiple files where components interact via a multitude of protocols and technologies. For example, SWQAT 1 assesses the coupling of modules (classes, files, etc.) via fan-in and fan-out of units, i.e., how often a unit is called by others and how often it is calling others. Yet, the metric is not assessed on higher levels of abstraction, such as components or systems. In contemporary software, these are usually the significant building blocks. Just like units, components can be strongly or loosely coupled. However, the static analysis rules of the SWQATs do not identify component interactions. These are difficult to assess automatically since they are often domain and technology specific. For example, components may be implemented in different programming languages using different technologies, where coupling may be introduced by inter-component calls via Remote Procedure Calls (RPC), via calls to REST APIs, etc. All of these are "invisible" to low-level fine-grained static analysis rules that operate on the ASTs of single-language artifacts. More expressive representations of software that capture cross-language and cross-component characteristics would be required to facilitate automatic checks on higher levels of abstraction. Similarly, interactions of components with e.g., databases cannot be assessed via low-level static analysis rules since these cross-language boundaries and since they are often hidden behind multiple technology-specific layers of abstraction.

Similarly, the SWQATs assess complexity on a very low level. Most often, complexity is assessed as the size of components measured in lines of code (LOC) or as the number of linearly independent paths through a program's source code (cyclomatic complexity). Such assessments may not always reflect code quality accurately as highly complex code is not necessarily bad if it's well-structured and well-documented. Furthermore, by only focusing on low-level forms of complexity, SWQATs are unaware of complexity problems that are more coarse-grained than the number of possible execution paths in units. More relevant problems that developers face revolve around the existence of test suites, to which degree they are readily executable, debuggability of systems, e.g., inspectability of running processes, executability of components in isolation, etc. Certainly, the number of technologies, programming languages, protocols, and components of a software system influences its complexity. Yet, such issues are not part of how SWQATs rate the complexity of a system.

The inability to assess a software systems' interaction between higher-level components can lead to deceptive ratings of important quality aspects. For example, software system 3 (Apache Airflow), which is a workflow orchestration engine that was originally developed by Airbnb [54], receives the worst "cloud readiness"-score from SWQAT 2 (see results in the reproduction kit [44]). This is quite a surprising result since this particular software system is primarily operating on Amazon Web Services (AWS) [55] and Google offers it as SaaS under the name Cloud Composer readily deployed to the Google Cloud Platform [56]. Results like this suggest that the provided rules encode a conception of quality, in this case, cloud readiness, that is not in line with reality.

Thus, our evaluation shows how contemporary SWQATs are unable to assess interactions and dependencies between higher-level components (written in multiple programming languages) that interact with each other via multiple protocols. As a result, SWQATs lack a holistic view of the software architecture and how different parts of the system interact, which is a major limitation in their ability to assess quality characteristics related to system integration, performance, maintenance, and security.

4.3 Challenge 3: Quality models and metrics vary greatly among different SWQATs

As listed in Table 4, SWQATs vary greatly in terms of what is considered to be software quality and how it is assessed. At first glance, the SWQATs resemble each other, as their quality models are often based on classical software quality models, such as the Boehm Model [14] or the McCall Model [57]. For example, SWQAT 1 implements the SIG/TÜViT Evaluation Criteria for Trusted Product Maintainability (SIG, 2020). This model consists of five dimensions—Analyzability, Modifiability, Testability, Modularity, and Reusability from ISO 25010—which are then aggregated into a single overall rating of 'Maintainability'. However, the exact procedure for how this aggregation is computed and which precise parameters are applied is not published. Similarly, for the other SWQATs 2 to 5, only partial descriptions of the proprietary software quality models are publicly available. SWQAT 6's quality model [58] is structurally similar to those of the other SWQATs, but it is the only one in this work that is completely traceable since it is implemented as an open-source tool [59] and extensively documented [60, 61].

Table 4: SWQATs' high-level model of software quality and number of metrics

SWQATs	**High-level Model of Software Quality**	**Number of metrics**
SWQAT 1	Proprietary, SIG/TÜViT Evaluation Criteria [63], inspired by ISO/IEC 25010	10 [36]
SWQAT 2	Proprietary, inspired by ISO/IEC 25010 and ISO/IEC 5055	Disclosed as "more than 325" [64]
SWQAT 3	Proprietary	Disclosed as "Multiple hundreds" [65]
SWQAT 4	Proprietary	10 [66]
SWQAT 5	Proprietary	10 [67]
SWQAT 6	SQALE method [58]	2.587 [44]

While the high-level models for software quality have similarities, the used metrics to vary greatly. Whereas SWQAT 1 relies on only 10 metrics, SWQAT 6 uses 2.587 metrics! Besides the quantitative difference, metrics differ also qualitatively. Metrics that are important in some SWQATs, are not considered to be important by others. A case in point is SWQAT 2's metric *"Javascript in HTML can be unreadable and cause reliability issues"* [62], which checks if an HTML file includes JavaScript code from an external file. None of the other SWQATs implements a corresponding metric or considers such a property to be detrimental to software quality.

Furthermore, while some metrics are common across multiple SWQATs, their implementation can still vary. For example, SWQATs 2, 4, 5, and 6 rely on a rule that checks that "*Classes should not have too many methods*" [68]. Still, the threshold for how many methods per class pose an issue to maintainability is different between the SWQATs. Per default, SWQAT 4 considers that at most 15 methods per class are appropriate [69], for SWQAT 5 it is at most 20 methods [70], for SWQAT 6 it is at most 35 methods [68], and for SWQAT 2 the precise number is undisclosed [71] and varies based on the respective version of the underlying regularly updated quality model. Note, this only illustrates the difference in thresholds that discern these rules.

Since most rules are proprietary, they cannot be inspected more closely to assess to which degree they identify the same source code constructs as methods, e.g., nested methods, methods with differing visibilities, etc. Previous work by Lincke et al. [72] demonstrates that differently implemented static analysis rules that assess the same metric lead to different and non-comparable results.

Similarly, while SWQAT 1, 2, 3, 4, and 6 all assess complexity in terms of cyclomatic complexity, they apply different thresholds. For instance, SWQAT 1 considers units with cyclomatic complexity above five detrimental to quality [36], for SWQAT 3 it is per default 20 [73], for SWQAT 4 and 6 it is 10 [74], and for SWQAT 2 the precise threshold depends on programming languages and is not disclosed [75]. Also, for cyclomatic complexity, it is unclear to which degree the implementations of the SWQATs differ. For example, for SWQAT 6 it is documented which keywords and logical operators of some programming languages are considered when estimating cyclomatic complexity [76]. However, it is unclear to which degree these keywords are suitable to estimate cyclomatic complexity or whether all SWQATs identify the same syntactic elements for the estimation of cyclomatic complexity.

The SWQATs in this evaluation do not only differ in the low-level metrics that they assess. Their quality models differ too, see Table 4. That is, the way in which the results of static analysis are aggregated into higher-level quality assessments is vendor-specific and different between SWQATs. For example, SWQAT 1 simply aggregates results in the form of a compliance ratio, i.e., how many of its ten quality metrics are not violated. For that, it relies on yearly recalibrated distributions of amounts of instances that violate a metric to a certain degree, e.g., at most 43.7% of all LOC are in units larger than 15 LOC, at most 25.2% of all LOC are in units with a cyclomatic complexity higher than 5, etc. [36]. SWQATs 2, 5, and 6 work similarly in terms of how frequencies of violations of static analysis rules are aggregated into intermediate scores, often via arithmetic means, threshold-based mappings, extreme value computations, etc. These intermediate scores are then further aggregated, again often just via computation of arithmetic means, and resulting scores are mapped to discrete ratings using a set of thresholds. These ratings are then presented by the SWQATs in their reports. In some cases, frequencies of violations of static analysis rules are not aggregated directly. For example, SWQAT 5 and 6 associate time estimates with static code analysis rules, which represent an estimate of how long it takes a developer to fix a violation of the corresponding rule (remediation time) normalized by an estimate of total system development time (based on LOC). The sum of remediation times is then mapped to discrete maintainability ratings [60, 67].

Our findings illustrate how contemporary SWQATs apply and aggregate quality metrics differently. Similar results have been reported in previous studies [58, 77]. This stresses the importance of knowing the differences in quality models to make informed choices about which SWQAT to apply in each project, how their results should be interpreted, and their reliability for decision-making. However, as showcased in this study, these insights into how the SWQAT differs are not always easy to access and in some cases, even disclosed.

4.4 Challenge 4: SWQAT ratings are incomparable and inconsistent

As a logical consequence of using different metrics, the resulting ratings are inconsistent and, practically, incomparable between SWQATs. As seen in Table 5, SWQAT results are phenomenologically incomparable since the ratings for various quality characteristics are reported on different scales and with different units. For example, the SWQATs in our study report results using nominal scales, absolute numbers, ratios, or frequencies of occurrences or violations of patterns. Overall quality and maintainability are reported mainly via discrete nominal scales.

However, the number of respective categories is different between SWQATs or the labels of categories are different. For example, SWQAT 2 reports overall quality, maintainability, and complexity on a three-step scale with values low, medium, and high, SWQAT 3 reports overall quality on a six-step scale with values A, B, C, D, E, and F, SWQAT 4 reports overall quality on a five-step scale with values A, B, C, D, and F, and SWQAT 6 reports maintainability on a five-step scale with values A, B, C, D, and E. SWQAT 1 reports maintainability not via a nominal scale but via a ratio of adherence to the ten metrics it uses for assessment. Except for the nominal scales with the same number of categories, it is not clear how reported values can be mapped to one another in a meaningful way.

Table 5: Overview of how the SWQATs rated the software quality of six software systems.

Quality characteristic	SWQAT	Metric name	Results of quality assessments per software system					
			Software system 1	Software system 2	Software system 3	Software system 4	Software system 5	Software system 6
Overall quality	SWQAT2	"Health"	Low	Medium	Medium	Medium	Medium	Low
	SWQAT3	"Software Quality Grade"	B	A	B	B	B	A
	SWQAT4	"Global project score"	n/a	B	A	A	A	B
Maintainability	SWQAT 1	"Maintainability Compliance"	n/a	6/10	4/10	4/10	7/10	5/10
	SWQAT 2	"Agility"	Medium	High	Medium	Medium	Medium	Low
	SWQAT 5	"Maintainability Grade"	C	n/a	C	C	C	D
	SWQAT 6	"Maintainability Rating"	A	A	A	A	A	A
Security	SWQAT 3	"Security"	710	0	291	192	0	0
	SWQAT 4	"Security"	n/a	0	0	0	0	0
	SWQAT 6	"Security Hotspots"	351	19	447	82	7	216
Complexity	SWQAT 2	"Elegance"	Low	Low	Low	Medium	Medium	Low
	SWQAT 3	"Complexity"	2%	0%	1%	0%	0%	1%
	SWQAT 4	"Complexity"	n/a	541	617	783	28	2.430
	SWQAT 5	"Complexity/Code smells"	14.514	n/a	1.479	773	17	1.296
	SWQAT 6	"Cyclomatic complexity", "Cognitive complexity"	124.876 113.650	4.392 2.945	18.057 11.881	13.243 7.768	666 306	86.414 249.097
Size	SWQAT 2	"LOC", "Files"	1.342.939 13.057	32.918 512	173.316 1.719	66.259 745	2.436 66	409.484 1.953
	SWQAT 3	"Total LOC"	1 M	41 K	190 K	137 K	3.097	94 K
	SWQAT 4	"Total LOC"	n/a	18.234	14.495	30.097	1.000	59.408
	SWQAT 5	"LOC", "Number of Files"	777.017 9.898	n/a n/a	116.171 1.735	81.846 1.254	1.167 39	71.354 608
	SWQAT 6	"LOC", "Files"	587.518 5.625	23.865 373	185.848 2.145	115.464 1.200	2.467 36	442.376 952
Duplication ("code clones")	SWQAT 3	"Duplication"	2%	3%	13%	11%	6%	n/a
	SWQAT 4	"Duplication"	n/a	65	34	97	2	221
	SWQAT 5	"Duplication"	6.491	n/a	360	699	6	478
	SWQAT 6	"Duplication density", "Duplication Files"	4.30% 703	5.20% 36	2.50% 146	2.40% 81	0% 0	71.80% 567

Even in the case of SWQATs using the same scales and units for reporting assessed quality characteristics, results are incomparable since the underlying metrics with associated static analysis rules differ between SWQATs. For example, SWQATs 3, 4, and 6 report assessments of the quality characteristic security via the total number of violations of static code analysis rules that scan code for potential security issues, see Table 5. However, the sets of security-related static analysis rules differ between the SWQATs. Of SWQAT 6's 2.587 static code analysis rules, 142 are security related, whereas SWQAT 3 relies on 58 rules that are enabled by default. The different number of metrics may explain the quite diverse numbers of potential security issues listed in Table 5.

Furthermore, quality assessments are inconsistent across the SWQATs since the results are neither proportional to each other nor are they ordered with respect to each other. Table 5 illustrates that maintainability is assessed constantly to be very good (A rating) for all software systems when assessed with SWQAT 6. However, when assessed with SWQAT 5, maintainability is almost constant on an average rating (C and D) and, when assessed with SWQATs 1 and 2, maintainability ratings vary more and in opposite directions. Software system 2 is assessed to be highly maintainable by SWQAT 2 but to be of average maintainability by SWQAT 1. System 5 receives an above-average maintainability rating from SWQAT 1 but an average maintainability rating (medium) from SWQAT 2, while SWQAT 2 rates system 6 to be of low maintainability whereas SWQAT 1 rates it as averagely maintainable. That is, maintainability ratings are transposed when compared across SWQATs. Another example of inconsistent assessment results can be found regarding security. Here, SWQAT 4 reports no security concerns in code in any of the six software systems (zero violations of static code analysis rules for the software system in Table 5). Hence, the software systems could be considered equally "secure". In contrast, SWQAT 3 finds security violations in software systems 1, 3, and 4, while SWQAT 6 reports security concerns in all of the six software systems. Assessments of the size of software systems are also inconsistent between SWQATs. The reported sizes in lines of code (LOC) differ by factors two to more than 12. For instance, SWQAT 4 reports that software system 3 consists of 14.495 LOC whereas SWQAT 3 reports it to consist of 190k LOC, see Table 5. Additionally, these factors are not proportional between SWQATs.

Finally, for the cloud-based SWQATs 1 to 5, assessment results are incomparable and inconsistent since they are not necessarily stable over time. The assessed quality of even an unchanged software system can change over time when vendors update the underlying quality models, their thresholds, etc. SWQAT users are not necessarily aware of these remote changes since such updates are not always communicated to users.

In sum, the reported results for the quality characteristics in Table 5 vary greatly across the SWQATs. What one SWQAT rates to be high quality, others rate as poor. This does not only highlight how SWQATs ratings are incomparable and inconsistent. It also highlights how SWQATs can produce false positives (identifying issues that are not actual problems) and false negatives (missing real issues). The presence of false results can affect the reliability of decision-making based on tool output.

4.5 Challenge 5: Results of quality assessments with SWQATs can be gamed

Because of the inconsistent ratings, software quality assessments can be gamed by choosing the "right" SWQAT. For example, three SWQATs (2, 3, and 4) aggregate the results of automatic quality assessments into a high-level quality grade, see the first row of Table 4. SWQAT 3 assesses the software systems 2 and 6 to be of top quality (grade A) while the remaining four systems are rated to be of good quality (grade B). SWQAT 4's quality grades appear to be inverted compared to those of SWQAT 3. SWQAT 2 assesses the software systems 2, 3, 4, and 5 to be of medium quality and the systems 1 and 6 to be of low quality. That means that users of SWQATs who are interested in a positive quality assessment of for example software system 6, can choose SWQAT 3 over SWQATs 4 or 2 to receive a "desired" result.

Similarly, users of SWQATs can "generate" favorable assessments for the other quality characteristics such as maintainability, security, complexity, size, and duplications (rows two to six in Table 4) by choosing the "right" tool. For example, the second row of Table 4 lists what SWQATs 1, 2, 5, and 6 assess to be the maintainability of the software systems in this evaluation. Where SWQAT 6 assesses all six systems to be excellently maintainable (A rating), SWQAT 2 only rates system 2 on that level of maintainability. According to SWQAT 2, software system 6 is of low maintainability and the remaining systems are of mediocre maintainability. SWQAT 5 assesses software systems 1, 3,

4, and 5 to be of mediocre maintainability and software system 6 to be of subpar maintainability. For SWQAT 1, software system 5 is the most maintainable (7/10) followed by system 2 (6/10). Software system 6 is rated mediocre (5/10) while systems 3 and 4 receive subpar ratings (4/10). Thus, a wide spread of maintainability ratings can be observed for software system 6. It ranges from an excellent A rating (SWQAT 6) over a mediocre 5/10 maintainability rating (SWQAT 1) and subpar D rating (SWQAT 5), to a worst low rating (SWQAT 2). For the other software systems, the maintainability assessments are more similar but still heterogeneous.

Following our findings, high – or low – ratings of quality in general or certain quality characteristics can be obtained by strategically choosing the "right" SWQATs for the assessment, something for IT owners, planners, and managers to keep in mind when they use SWQAT ratings to support decisions regarding contract negotiations, project terminations, or settling disputes.

5. Towards more tailorable SWQATs

In response to the challenges of contemporary SWQATs identified above, we present five design requirements and a high-level design proposal that allows for more conscious and prudent software quality assessments that better reflect the socio-technical aspect of software systems and the context-specific nature of software quality. For each challenge, we discuss the underlying issue and formulate a requirement that our design for a SWQAT has to support. Finally, we illustrate our design and map the distilled requirements to design decisions.

5.1 Design requirements

The first challenge asserts that SWQATs assess software quality quite narrowly as certain low-level aspects of source code quality. However, software quality is usually more than source code quality. Industrial standards like the ISO/IEC 25000 standard define software quality more broadly as "the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs" [26], which includes quality in use, data quality, etc. Accordingly, machine-readable artifacts like configurations, data, natural language documents, etc. that are often considered key parts of the software too, should be included in assessments related to, e.g., maintainability [51, 78]. This leads us to the first design requirement:

> ***Design requirement 1:*** *The SWQAT should be able to assess quality of more machine-readable artifacts than just source code.*

The SWQATs in our evaluation assess software quality via static analysis of source code separately over files, see Challenge 2. This neglects runtime behavior that has an impact on software quality too. For example, one can think of a distributed software system with formidable source code quality, which may be unusable due to low quality of network connections, an aspect that is invisible in source code and that first can be revealed via dynamic analysis. Additionally, other industrial standards like ISO/IEC 9001 define software quality as the "capability of a software product to conform to requirements" [79] as in ISO/IEC 9001. Conformance to requirements cannot be assessed with static analysis but needs more advanced techniques, such as dynamic analysis in combination with user studies [80]. Consequently, we can identify the second requirement for the design:

> ***Design requirement 2:*** *The SWQAT can ingest results from a wide variety of metrics that measure inputs from sources ranging from machine-readable artifacts to knowledge carried by humans.*

Challenge 3 highlighted that there is significant variation in the software quality characteristics and metrics used by different SWQATs. This inconsistency arises from the fact that SWQATs come with predefined vendor-specific metrics and static analysis rules that cannot be customized beyond merely adjusting various thresholds. This limitation means that users are often unable to tailor the metrics and rules to meet their specific needs, which may result in unreliable or irrelevant results.

Furthermore, the utility and appropriateness of some of the metrics that contemporary SWQATs implement are highly debatable. For example, many SWQATs assess complexity via cyclomatic complexity even though the metric is heavily criticized for its lack of expressiveness and theoretical foundations [81]. To mitigate Challenge 3, we arrive at the third design requirement:

> ***Design requirement 3****: The SWQAT has to support the user in defining and customizing the metrics used to measure software quality. Additionally, it should support the user to precisely specify measurement targets and approaches.*

The results of our evaluation demonstrate that the quality assessments of SWQATs are inconsistent and incomparable, see Challenge 4. The main reason for that issue is that vendors provide predefined, vendor-specific, and non-tailorable quality models with their tools. That is, contemporary SWQATs encode an assumption that software quality can be uniformly assessed over different business domains and types of software. This contradicts extant research that describes software quality to be different depending on domain and perspective [20], and also contradicts industrial standards like the ISO/IEC 25000 series of standards [26] or the SQALE method [25] that suggest to tailor the provided quality model before assessing software quality. Consequently, we describe a fourth design requirement to improve the reliability of SWQATs:

> ***Design requirement 4:*** *The SWQAT has to support the user in defining what software quality is and in customizing the underlying quality model accordingly.*

Finally, Challenge 5 describes that results of quality assessments with SWQATs can be gamed. There are two underlying reasons for this. First, as covered in Challenges 3 and 4, the inconsistencies across SWQAT ratings allow users to obtain different results depending on the tool they use. Secondly, SWQATs lack transparency on how they define software quality and how their assessments, results, and reports are generated. Often, the underlying quality models are only described vaguely in the documentation and the precise metric implementations are rarely publicly available. To mitigate the possibility of gaming results, we describe the fifth requirement as follows.

> ***Design requirement 5:*** *The SWQAT has to be completely transparent in what it considers to be software quality, how it assesses software quality, and how quality ratings, assessments, visualizations, or reports are created.*

5.2 Design proposal

Based on the five requirements above, we propose a high-level design for tailorable SWQATs. Figure 1 illustrates our design as Unified Modeling Language (UML) component diagram. Conceptually, the design consists of three components: The Software Quality Model component that contains an encoded definition of what characterizes software quality for the specific context, the Metric component that implements how certain quality characteristics should be measured, and the Report component that transparently converts the results into suitable presentations.

The Metric component captures both machine-readable artifacts (e.g., source code or documentation) and expert knowledge (e.g., the architecture of a system, deployment processes, or information about how to handle errors in production). In case of measurement of expert knowledge, either that knowledge is transferred into machine-readable documents for which a metric component is implemented or the results of a manual measurement are encoded in a metric component directly. After a quality assessment (time aspect is not illustrated in Figure 1), in which the Software Quality Model ingests all measurement results from the Metric component, it exposes all these and aggregated quality assessments to the Report component, which in turn transparently presents the results as raw data (e.g., the number of violations and occurrences of violations of a static analysis rule implemented in a metric component) or more aggregated data (e.g., overall ratings or visualizations).

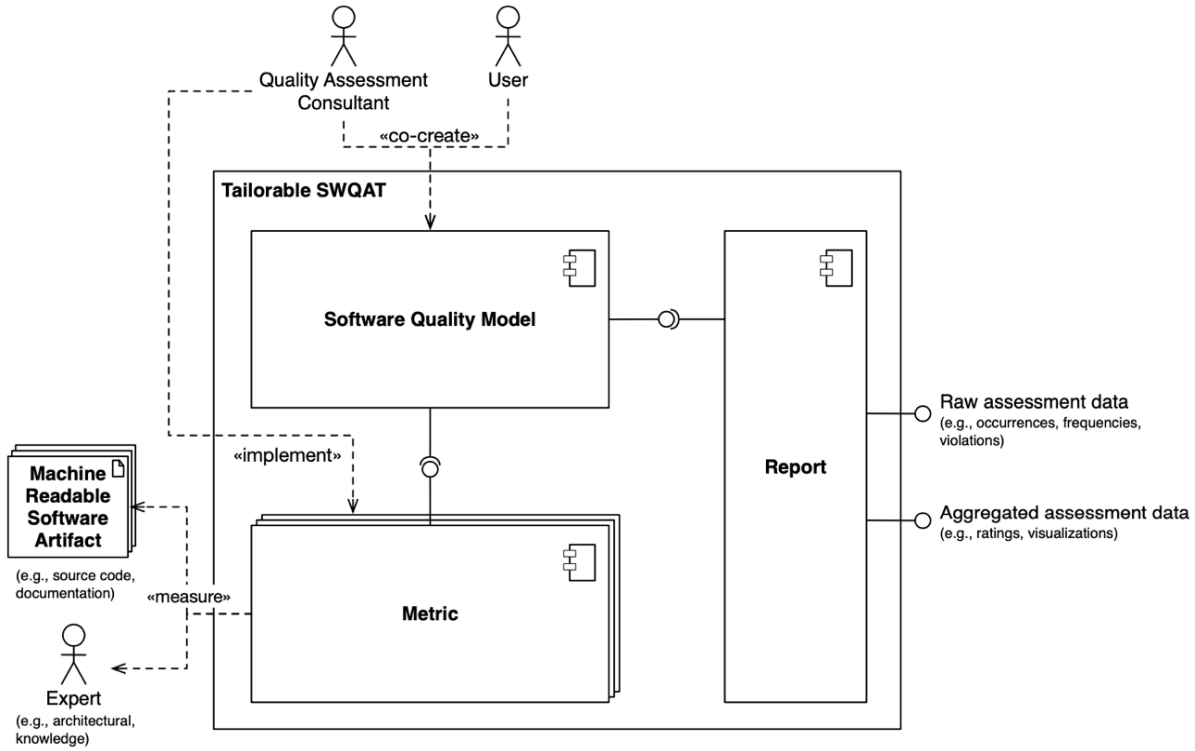

Figure 1: Design proposal for a tailorable SWQAT.

The design denotes three distinct roles or functions that are necessary for the effective functioning of the SWQAT. These roles are referred to as User, Quality Assessment Consultant, and Expert. The User represents the organization that wants to assess the quality of specific software systems. This could be an IT manager or any other person responsible for overseeing the quality of these systems. The User's primary responsibility is to provide input on the software quality requirements, objectives, and constraints to the Quality Assessment Consultant. That is, together with the Quality Assessment Consultant, the User tailors the software quality model. The Quality Assessment Consultant is responsible for encoding the software quality model, implementing respective metrics, and applying the SWQAT to the software systems that are assessed. The consultant can be an employee of the organization itself or an external consultant hired specifically for the purpose of assessing the software quality. The consultant works closely with the user to ensure that the software quality assessment meets the specific requirements, objectives, and constraints of the organization. The third role, the Expert, represents people who have insights into the assessed software and bear knowledge that supplements source code, e.g., on undocumented aspects that are relevant to the assessed software. The Experts' role is to provide data for metrics that are not captured in the machine-readable artifacts.

Overall, the proposed design emphasizes the importance of involving different roles with specific expertise in software quality assessment to ensure a comprehensive and accurate evaluation of the software system. This process is carried out in three steps.

First, the User and Quality Assessment Consultant co-create the software quality model that serves as a reference system for the assessments. This task involves defining what software quality means in the specific context and deciding which qualities shall be assessed and how they shall be measured. By involving both the user and consultant in this step, the resulting quality model reflects the context of the specific software systems that are assessed, addressing design requirements 3 and 4.

Next, the Quality Assessment Consultant implements metrics that either automatically assess certain qualities of software artifacts or wrap the results of (semi-)automatic assessments of certain qualities. This includes input measures from machines and experts, enabling the assessment of qualities that cannot be measured automatically. By doing so, the proposed design addresses design requirements 1 and 2, and expands the assessment beyond just source code in text files.

Finally, the SWQAT converts the results transparently into reports, visualizations, or raw assessment data that is suitable for the user. Transparency is achieved through the active involvement of the user in the specification of what is considered software quality in this context, how software quality is measured, and how the ratings, assessments, visualizations, or reports are created. In addition to ensuring that the SWQAT report is relevant to the specific context, the active involvement also makes the quality assessment process traceable and visible to the user. Thereby, the proposed design addresses design requirement 5.

Being deliberately high-level, our design proposal can be applied differently, depending on project development methods (e.g., agile vs. plan-driven), primary users (e.g., internally by the project team during development or externally by consultants to assess project deliveries), and at various stages of the product life-cycle (e.g., during development or after implementation). For instance, when used by a project team relying on agile software development methods, the tailorability of both metrics and software quality model allows for iterative and incremental assessment of software quality during development. Per development cycle, metrics and quality model may change to reflect changes in scope or requirements following a development iteration [28]. Thereby, quality assessments can evolve together with the software system under development, similar to associated test suites. In plan-driven development metrics and software quality model can be specified up-front, e.g., during contract negotiations with suppliers [see 9,10], to serve as a target that a developed software product has to reach before project end. Similarly, if quality assessments are used for settling legal disputes between suppliers and clients [see 13], contracts and project documentation can serve as basis for tailoring a SWQAT to ensure a more nuanced, transparent, and context-dependent assessment than what contemporary off-the-shelf tools can offer, minimizing the risk of gaming (i.e., selecting a specific SWQAT to ensure a certain outcome).

Taken together, the proposed design allows for a more conscious and prudent application of SWQATs, enabling a comprehensive and accurate evaluation of software systems that better reflect their socio-technical nature. By involving different roles with specific expertise in the software quality assessment process and by tailoring the SWQAT to specific contexts, our design provides a flexible and adaptable solution for assessing software quality. Thus, users in our design are considered broadly to be any stakeholder, including project managers, developer teams, IT departments, lawyers, consultants, etc., that want to assess the quality of specific software systems. With context-specific quality models and metrics, they can use a tailorable SWQAT to assess if a software system is ready for delivery, where to focus maintenance efforts, determine if the quality of the system aligns with the specifications outlined in the contract, and so on. Since the practical implementation of metrics and quality models into a tailorable SWQAT is a technical task that requires programming skills, users in any kind of usage scenario are accompanied by quality assessment consultants that translate a user's software quality concerns into a suitable format for the SWQAT. Thereby, in contrast to the predefined, one-size-fits-all approaches embedded in contemporary SWQATs, the tailorable SWQAT design encourages collaboration between different actors to better reflect software quality as a context-dependent and socio-technical phenomenon.

6. Discussion

The aim of this paper is to advance knowledge on the capabilities and limitations of contemporary SWQATs in information systems development and maintenance, specifically by identifying key challenges and developing a design proposal to address these challenges. Below, we discuss the implications of our results and proposed design before outlining the limitations of our work and suggestions for future research.

6.1 Implications for research

Our research has two implications for research. First, to our knowledge, no prior research compares and critically evaluates how SWQATs assess various software systems. In this regard, the results in this study presents a novel contribution by highlighting several risks with using SWQAT ratings at face value as a basis for decision-making in information systems development and maintenance. As such, our research challenges the predominantly technical-oriented and optimistic literature on SWQATs, which, until now, emphasizes the value of contemporary tools for improving and managing the technical quality of software systems [3, 5, 7, 30]. Since software is a socio-technical artifact and software quality encompasses more than just internal quality characteristics of source code [20], it is very likely that purely automatic assessment of software quality is not practicable, as reflected in the challenges identified in this study. Our design offers an initial conceptual framework for semi-automatic assessments that also includes machine-readable documents other than source code, stakeholder surveys, and automatic measurement of metrics.

Second, although SWQATs receive increasing attention from IT developers, managers, and consultants in practice, research on SWQATs has largely remained within the computer science community regarding software quality metrics [2, 6] or how to develop different types of quality assessment tools [5, 30]. However, as these tools increasingly impact decision-making in various aspects of information system development and maintenance, it is crucial to explore this topic from multiple disciplinary perspectives. Here, IS researchers are uniquely positioned to contribute theory on how SWQATs can be used prudently in different contexts, how various stakeholders understand and apply them, and how SWQAT ratings influence decision-making throughout IT product lifecycles. By highlighting five key challenges to the reliability of using SWQATs in IT management, this article paves the way for new research avenues for understanding the complexities of assessing and ensuring high quality as software systems grow in numbers and sophistication.

6.2 Implications for practice

Since contemporary SWQATs are based on predefined, non-tailorable, and non-transparent software quality models, users of these tools must subscribe to the given vendor's understanding of software quality and trust that the metrics applied are relevant to the specific context. To increase the transparency and reliability of SWQATs, vendors should aim to make metrics, static analysis rules, and aggregation mechanisms as accessible and understandable as possible. To achieve this, SWQAT vendors can incorporate our design proposal to make their SWQATs more tailorable.

It is important to emphasize that our proposed design for tailorable SWQATs does not imply that SWQAT vendors and users should discard their existing quality models and start from scratch with each quality assessment. Instead, we suggest that–in line with industrial software quality standards– for most cases, using a standard quality model as a starting point and then adapting it to the specific context would be a useful approach. This allows for a more efficient and effective tailoring of SWQATs, while still building on established and widely accepted quality models. Furthermore, we envision that the components of tailorable SWQATs can be distributed as open-source, so that software quality assessment consultants can reuse and share previous software quality models and corresponding metrics. A catalog of reusable metrics and exemplary quality models is likely to increase adoption of tailorable SWQATs and make future quality assessments more efficient.

Our suggested design allows for incorporation of human knowledge for software quality assessments. However, the more non-automatic assessments are required, the more cumbersome and resource-demanding it becomes to continuously assess software quality. While a more participatory process initially might be more resource-intensive for the user than procuring and applying non-configurable off-the-shelf SWQATs, we believe the investment will pay off

by generating relevant, transparent, and more reliable assessments to inform decision-making. If tailoring a software quality model is not possible due to a lack of expertise or lack of resources, we recommend applying more than one readily available SWQAT at a time and considering the respective results in light of the challenges identified in our evaluation.

6.3 Limitations and future research

Our work has several limitations. First, we conducted a technical evaluation [35] for which we selected six SWQATs to assess the quality of six software systems. It is important to note that the landscape of available SWQATs is constantly changing with ongoing updates to current tools and the launch of new ones. While this dynamic aspect of SWQATs provides limitations to the generalizability of our results, it simultaneously reflects the growing use of these tools in practice and stresses the importance of researching issues related to how they function, what they measure, and how suitable they are for decision-making. Second, we conduct a purely technical evaluation and not a field study. Practitioners may be aware of the shortcomings, inner workings, inconsistencies, etc., between SWQATs when applying them in practice and circumvent these accordingly. However, from our professional experience, we believe this is rarely the case. In the future, we plan to study how SWQATs are applied in industrial practice, which expectations users have towards the results of SWQATs, and how results of automatic quality assessments are operationalized. Finally, further research is needed to evaluate the practical feasibility and potential of our high-level design proposal.

7. Conclusion

Software quality is a fundamental aspect of information systems development and maintenance. It not only ensures the reliability, security, and efficiency of information systems but also plays a crucial role in driving stakeholder satisfaction, cost-effectiveness, and the long-term success of the systems being developed. Organizations increasingly turn to software quality assessment tools (SWQAT) as a new way to manage and automatically assess the technical state of the ever-growing number of large information systems throughout their life cycle. In this paper, we evaluated the reliability of SWQATs for decision-making and proposed a design for how reliability can be improved. Our findings highlighted five key challenges of using software quality assessment tools. First, contemporary SWQATs are based on a narrow and one-dimensional conceptualization of software quality that strictly focuses on source code and leaves out other important aspects, such as the degree to which the systems meet the functional requirements, data quality, or intended usage. Second, SWQATs are based on low-level static analysis of single files and languages and disregard that software systems are often constructed out of many interacting components (written in multiple programming languages) that interact with each other via a plethora of protocols. Third, SWQATs vary considerably with regard to their conception and assessment of software quality. Fourth, software quality assessments are diverse, incomparable, and inconsistent. Five, because of the inconsistent ratings, results of software quality assessments can be gamed by choosing the “right” SWQAT. Taken together, our results cast doubt on the suitability of relying exclusively on SWQATs for decision-making, something for IT owners, planners, and managers to keep in mind when they use SWQAT ratings to support decisions regarding contract negotiations, project terminations, or settling disputes.

To overcome these challenges, we have proposed a high-level design for tailorable SWQATs that enables a more conscious and context-specific application of these tools by more actively involving the user and incorporating a wider variety of metrics that measure inputs from sources ranging from machine-readable artifacts to knowledge carried by humans. Thereby, the design is aimed to better reflect the socio-technical nature of software systems. By allowing users to tailor the assessment criteria and metrics to their specific needs and circumstances, such SWQATs would provide more relevant and meaningful insights to inform decisions about information systems development and maintenance.

References

[1] CAST (2014, December 5). *Coca-Cola Prevents Operations Failures with CAST* [Online]. Available: http://youtu.be/HheIYXhTLQ4&list=PL67ABA980FDDBB909&index=50

[2] N. Gorla and S.-C. Lin, "Determinants of software quality: A survey of information systems project managers," *Information and Software Technology*, vol. 52, no. 6, pp. 602–610, 2010.

[3] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[4] V. S. Bhadauria, R. K. Mahapatra, and S. P. Nerur, "Performance outcomes of test-driven development: An experimental investigation," *Journal of the Association for Information Systems*, vol. 21, no. 4, Article 2, 2020.

[5] M. Rodriguez, M. Piattini, and C. Ebert, "Software verification and validation technologies and tools," *IEEE Software*, vol. 36, no. 2, pp. 13–24, 2019.

[6] M. Nilson, V. Antinyan, and L. Gren, "Do internal software quality tools measure validated metrics?" in *20th International Conference on Product-Focused Software Process Improvement (PROFES),* Barcelona, Spain, 2019, pp. 637–648.

[7] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. A. Fontana, T. Besker, A. Chatzigeorgiou, et al., "An overview and comparison of technical debt measurement tools," *IEEE Software*, vol. 38, no. 3, pp. 61–71, 2020.

[8] C. Zhi, S. Deng, J. Yin, M. Fu, H. Zhu, Y. Li, and T. Xie, "Quality assessment for large-scale industrial software systems: Experience report at Alibaba," in *26th Asia-Pacific Software Engineering Conference (APSEC)*, Putrajaya, Malaysia, 2019, pp. 142–149.

[9] KOMBIT (2015, May 19). *Social Pension Kommune Bilag 02.01 Kravspecifikation* [Online]. Available: https://www.kombit.dk/sites/default/files/user_upload/documents/Social_Pension/Bilag%202.1%20Kravspecifikation%20Leverand%C3%B8rh%C3%B8ring.pdf

[10] Kromann Reumert (2021, June 23). *Kodekvalitet i IT-kontrakter* [Online]. Available: https://kromannreumert.com/sites/kromannreumert.com/files/media/document/21x21%20Brochure%20Kodekvalitet%20i%20kontrakter.pdf

[11] Danish Parliament (2013, May 23). *Aktstykke nr. 127* [Online]. Available: https://www.ft.dk/RIpdf/samling/20121/aktstykke/aktstk127/20121 _aktstk_afgjort127.pdf

[12] Danish Parliament (2012, June 7). *Aktstykke nr. 167* [Online]. Available: https://www.ft.dk/RIPdf/samling/20142/aktstykke/aktstk167/20142 _aktstk_afgjort167.pdf

[13] The Supreme Court of Denmark (2013, April 25). *Forsvarets Materieltjeneste mod Saab AB* [Online]. Available: http://91.229.236.31/~itkontraktret/wp-content/uploads/2020/08/H-2013-04-25-Forsvarets-Materieltjeneste-mod-Saab-AB.pdf

[14] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in Proceedings of the *2nd International Conference on Software Engineering*, San Francisco, USA, 1976, pp. 592–605.

[15] V. Jayakody and J. Wijayanayake, "Critical success factors for DevOps adoption in information systems development," *International Journal of Information Systems and Project Management*, vol. 11, no. 3, pp. 60-82, 2023.

[16] P. Kapur, A. Gupta, P. Jha, and S. Goyal, "Software quality assurance using software reliability growth modelling: state of the art," *International Journal of Business Information Systems*, vol. 6, no. 4, pp. 463–496, 2010.

[17] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Addison-Wesley Professional, 2011.

[18] D. S. Guaman, J. M. Del Alamo, and J. C. Caiza, "A systematic mapping study on software quality control techniques for assessing privacy in information systems," *IEEE Access*, vol. 8, pp. 74808–74833, 2020.

[19] S. Khaddaj and G. Horgan, "The evaluation of software quality factors in very large information systems," *Electronic Journal of Information Systems Evaluation*, vol. 7, no. 1, pp. 43–48, 2004.

[20] B. Kitchenham and S. L. Pfleeger, "Software quality: The elusive target [special issues section]," *IEEE Software*, vol. 13, no. 1, pp. 12–21, 1996.

[21] L. A. Von Hellens, "Information systems quality versus software quality: a discussion from a managerial, an organisational and an engineering viewpoint," *Information and Software Technology*, vol. 39, no. 12, pp. 801–808, 1997.

[22] M. Jørgensen, "Software quality measurement," *Advances in Engineering Software*, vol. 30, no. 12, pp. 907–912, 1999.

[23] Y. H. Yang, "Software quality management and ISO 9000 implementation," *Industrial Management & Data Systems*, vol. 101, no. 7, pp. 329-338, 2001.

[24] G. H. Subramanian, J. J. Jiang, and G. Klein, "Software quality and is project performance improvements from software development process maturity and is implementation strategies," *Journal of Systems and Software*, vol. 80, no. 4, pp. 616–627, 2007.

[25] J.-L. Letouzey and M. Ilkiewicz, "Managing technical debt with the SQALE method," *IEEE Software*, vol. 29, no. 6, pp. 44–51, 2012.

[26] ISO Central Secretary (2011). *Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models (Standard No. ISO/IEC 25010:2011)* [Online]. Available: https://www.iso.org/standard/35733.html

[27] J. S. Saltz, K. Crowston, R. Heckman, and Y. Hegde, "MIDST: an enhanced development environment that improves the maintainability of a data science analysis," *International Journal of Information Systems and Project Management*, vol. 8, no. 3, pp. 5-22, 2020.

[28] M. Kuciapski and B. Marcinkowski, "Agile software development approach for 'ad-hoc' IT projects," *International Journal of Information Systems and Project Management*, vol. 11, no. 4, pp. 28-51, 2023.

[29] D. Galin, *Software Quality Assurance: From Theory to Implementation*. Pearson education, 2004.

[30] G. A. Campbell and P. P. Papapetrou, *Sonarqube in Action*. Manning Publications Co., 2013.

[31] SIG (2022, Nov. 8). *We are Software Improvement Group*. [Online]. Available: https://www.softwareimprovementgroup.com/about/

[32] Danish Agency for Digital Government. (2021, June). *Vejledning til model for porteføljestyring af statslige it-systemer*. [Online]. Available: https://digst.dk/media/21352/vejledning-til-model-for-portefoeljestyring-af-statslige-it-systemer1-kopi.pdf

[33] R.-H. Pfeiffer. "The Impact of Continuous Code Quality Assessment on Defects," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Luxembourg, 2021, pp. 624–628.

[34] diN. H. Thuan, A. Drechsler, and P. Antunes, "Construction of design science research questions," *Communications of the Association for Information Systems*, vol. 44, no. 1, p. 20, 2019.

[35] J. Venable, J. Pries-Heje, and R. Baskerville, "FEDS: a framework for evaluation in design science research," *European Journal of Information Systems*, vol. 25, p. 77–89, 2016.

[36] J. Visser, S. Rigal, and R. van der Leek. *Building maintainable software, java edition*, O'Reilly Media, 2016.

[37] CAST (2022, October 2). *What makes CAST technology so unique* [Online]. Available: https://web.archive.org/web/20221002024920/https://www.castsoftware.com/Overview/why-cast

[38] BetterCodeHub (2022, August 13). *Better Code Hub configuration files* [Online]. Available: https://web.archive.org/web/20220813060825/https://bettercodehub.com/docs/configuration-manual

[39] CAST. *Product Tutorials & Third-Party Tools.* [Online]. Available: https://doc.casthighlight.com

[40] Codacy. *Documentation home* [Online]. Available:https://docs.codacy.com

[41] Codebeat. *What languages does codebeat support?* [Online]. Available: https://hub.codebeat.co/docs/

[42] CodeClimate. *Adding Your First Repo* [Online]. Available: https://docs.codeclimate.com/docs/

[43] Sonarqube (2022, September 6). *Documentation* [Online]. Available: https://web.archive.org/web/20221123090625/http://docs.sonarqube.org/7.9/

[44] R.-H. Pfeiffer. *Reproduction kit* [Online]. Available: https://zenodo.org/records/10556441

[45] Apache Ignite. *Distributed Database For HighPerformance Applications With InMemory Speed* [Online]. Available: https://ignite.apache.org

[46] Apache Commons. *Commons Virtual File System* [Online]. Available: https://commons.apache.org/proper/commons-vfs

[47] Apache Airflow. *Apache Airflow* [Online]. Available: https://airflow.apache.org

[48] Apache Superset. *Apache Superset* [Online]. Available: https://superset.apache.org

[49] Apache Cordova. *Apache Cordova* [Online]. Available: http://cordova.apache.org

[50] Apache ECharts. *Apache ECharts* [Online]. Available: https://echarts.apache.org

[51] R.-H. Pfeiffer. "What constitutes Software?: An Empirical, Descriptive Study of Artifacts," in Proceedings of the *17th International Conference on Mining Software Repositories*, Seoul, Republic of Korea, 2020, pp. 481–491.

[52] CAST. *Switch cases without ending breaks are hard to understand* [Online]. Available: https://doc.casthighlight.com/alt_missingbreakincasepath-the-code-contains-too-many-switch-cases-with-missing-ending-breaks

[53] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*, Pearson Education, 2002.

[54] Amazon Web Services. *Airbnb on AWS* [Online]. Available: https://aws.amazon.com/solutions/case-studies/airbnb/

[55] Amazon Web Services. *Amazon Managed Workflows for Apache Airflow* [Online]. Available: https://aws.amazon.com/managed-workflows-for-apache-airflow/

[56] Google Cloud. *Cloud Composer overview* [Online]. Available: https://cloud.google.com/composer/docs/concepts/overview

[57] J. A. Cavano and J. A. McCall, "A framework for the measurement of software quality," *ACM SIGSOFT Software Engineering Notes*, Volume 3, Issue 5, pp 133–139, 1978.

[58] J.-L. Letouzey (2016). *The SQALE method for managing technical debt definition document* [Online]. Available: http:// www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf.

[59] GitHub. *Sonarqube* [Online]. Available: https://github.com/SonarSource/sonarqube

[60] Sonarqube. *Metric definitions* [Online]. Available: https://docs.sonarqube.org/latest/user-guide/metric-definitions/

[61] Sonarqube (2022, March 2). *Adding Coding Rules* [Online]. Available: https://web.archive.org/web/20220302070056/https://docs.sonarqube.org/7.8/extend/adding-coding-rules/

[62] CAST. *JavaScript in HTML can be unreadable and cause reliability issues* [Online]. Available: https://doc.casthighlight.com/alt_include-avoid-include-javascript-files/

[63] SIG. (2020*). SIG/TÜViT Evaluation Criteria Trusted Product Maintainability, Version 12.0*. [Online]. Available: https://www.softwareimprovementgroup.com/wp-content/uploads/2020-SIG-TUViT-Evaluation-Criteria-Trusted-Product-Maintainability.pdf.

[64] CAST. *Indicators Methodology* [Online]. Available: https://doc.casthighlight.com/category/product/indicators-methodology

[65] Codacy. *Configuring code patterns* [Online]. Available: https://docs.codacy.com/repositories-configure/configuring-code-patterns

[66] Codebeat. *Metrics customization* [Online]. Available: https://hub.codebeat.co/docs/metrics-customization

[67] CodeClimate. *Our 10-Point Technical Debt Assessment* [Online]. Available: https://codeclimate.com/blog/10-point-technical-debt-assessment

[68] GitHub. *TooManyMethodsCheck.java* [Online]. Available: https://github.com/SonarSource/sonar-java/blob/62670ebc03aa01346f96d40a2aa999db3487d973/java-checks/src/main/java/org/sonar/java/checks/TooManyMethodsCheck.java

[69] Codebeat. *Number of functions* [Online]. Available: https://hub.codebeat.co/docs/namespace-level-metrics#number-of-functions

[70] CodeClimate. *Sample .codeclimate.yml* [Online]. Available: https://docs.codeclimate.com/docs/default-analysis-configuration#sample-codeclimateyml

[71] CAST. *Avoid files with too many functions or methods* [Online]. Available: https://doc.casthighlight.com/alt_nbmeth_avoid-files-with-too-many-functions-or-methods/

[72] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, USA, pp. 131–142, 2008.

[73] Codacy. *Which metrics does Codacy calculate?* [Online]. Available: https://docs.codacy.com/faq/code-analysis/how-does-codacy-measure-complexity-in-my-repository/

[74] Codebeat. *Function-level metrics* [Online]. Available: https://hub.codebeat.co/docs/software-quality-metrics

[75] CAST. Structural code complexity may be too high [Online]. Available: https://doc.casthighlight.com/alt_vg-structural-code-complexity-may-high/

[76] Sonarqube (2020, September 2). *Metric Definitions* [Online]. Available: https://web.archive.org/web/20200927210528/https://docs.sonarqube.org/7.9/user-guide/metric-definitions/

[77] Pizzutillo, P. (2020). *CAST Highlight Metrics & Methodology*. [Online]. Available: https://doc.casthighlight.com/tools/CAST-Highlight-Indicators-Methodology.pptx

[78] L. J. Osterweil, "What is software?," *Automated Software Engineering*, vol. 15, no. 3, pp. 261–273, 2008.

[79] ISO Central Secretary. (2015). *Quality management systems – requirements (Standard No. ISO/IEC 9001:2015). Geneva, CH: International Organization for Standardization*. [Online]. Available: https://www.iso.org/standard/62085.html

[80] S. Lauesen and H. Younessi, "Is software quality visible in the code?," *IEEE Software*, vol. 15, no. 4, pp. 69–73, 1998.

[81] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, 1988.

Biographical notes

Rolf-Helge Pfeiffer
Rolf-Helge Pfeiffer is an associate professor at the IT University of Copenhagen in the Research Center for Government IT. His research interests and teaching are in the areas of software engineering, software quality, software quality metrics, and technical debt. Prior to his academic career, he worked as a software engineer and team lead at the Danish Meteorological Institute, where he was responsible for development and maintenance of 24/7 remote sensing software systems.

Jon Aaen
Jon Aaen is a postdoc at the Danish Institute for IT Program Management, IT University of Copenhagen. He obtained his Ph.D. at the Center for Information Systems Management, Aalborg University, Denmark. His research focuses on organizational and managerial aspects of IT project management and digital innovation. His work has been published in several scholarly journals, including the European Journal of Information Systems, the Scandinavian Journal of Information Systems, and Information Polity.