Empirical Software Engineering (2024) 29:53
https://doi.org/10.1007/s10664-023-10390-z

Investigating the readability of test code

Dietmar Winkler[1,2,3] · **Pirmin Urbanke**[4] · **Rudolf Ramler**[5]

Accepted: 30 August 2023 / Published online: 26 February 2024

Abstract

Context The readability of source code is key for understanding and maintaining software systems and tests. Although several studies investigate the readability of source code, there is limited research specifically on the readability of test code and related influence factors.

Objective In this paper, we aim at investigating the factors that influence the readability of test code from an academic perspective based on scientific literature sources and complemented by practical views, as discussed in grey literature.

Methods First, we perform a Systematic Mapping Study (SMS) with a focus on scientific literature. Second, we extend this study by reviewing grey literature sources for practical aspects on test code readability and understandability. Finally, we conduct a controlled experiment on the readability of a selected set of test cases to collect additional knowledge on influence factors discussed in practice.

Results The result set of the SMS includes 19 primary studies from the scientific literature for further analysis. The grey literature search reveals 62 sources for information on test code readability. Based on an analysis of these sources, we identified a combined set of 14 factors that influence the readability of test code. 7 of these factors were found in scientific *and* grey literature, while some factors were mainly discussed in academia (2) *or* industry (5) with only limited overlap. The controlled experiment on practically relevant influence factors showed that the investigated factors have a significant impact on readability for half of the selected test cases.

Conclusion Our review of scientific and grey literature showed that test code readability is of interest for academia and industry with a consensus on key influence factors. However, we also found factors only discussed by practitioners. For some of these factors we were able to confirm an impact on readability in a first experiment. Therefore, we see the need to bring together academic and industry viewpoints to achieve a common view on the readability of software test code.

Communicated by: Simone Scalabrino, Rocco Oliveto, Felipe Ebert, Fernanda Madeiral, Fernando Castor

This article belongs to the Topical Collection: *Special Issue on Code Legibility, Readability, and Understandability*.

✉ Dietmar Winkler
dietmar.winkler@tuwien.ac.at

Extended author information available on the last page of the article

Keywords Test code · Readability · Understandability · Maintainability · Systematic mapping study · Grey literature · Controlled experiment

1 Introduction

Software Tests encode important knowledge about typical usage scenarios and inputs, corner cases, exceptional situations as well as the intended output and behavior of the software system. Consequently, test cases play an important role for assuring software quality and also for the evolution of software systems as a knowledge source that supports communication in the team and with customers Latorre (2014) and for specification and documentation (Ricca et al. 2009). At the same time, the evolution of software systems requires that test cases are frequently updated and extended, resulting in effort for corresponding test case evolution and maintenance activities (Moonen et al. 2008; Zaidman et al. 2011).

Test Code Garousi and Felderer (2016), i.e., the form in which executable automated tests are commonly available, is the basis for many downstream activities such as maintaining and refactoring tests, locating faults, debugging, analyzing and comprehending test results, repairing broken tests, or dealing with flakiness (Garousi and Felderer 2016). In all these scenarios, developers and testers have to repeatedly read and understand test code – a usually time-consuming manual task, which makes readability and understandability critical factors when it comes to the quality of a project's test cases (Kochhar et al. 2019; Setiani et al. 2021a).

Readability as well as legibility and understandability of source code have already been subject to a series of empirical studies, which were recently examined in a systematic literature review by Oliveira et al. (2020). Test code has many properties in common with source code of software programs, and tests are often written using the same programming languages as the system under test. Nevertheless, the development of test code also shows significant differences when compared to other code. There exist dedicated frameworks and patterns for implementing test code (Meszaros 2007) and, furthermore, tools for generating tests (e.g., Evosuite, Randoop, IntelliTest) are becoming increasingly popular (Ramler et al. 2018).

In this paper, we focus on **investigating the readability of software test code by combining scientific and practical views**.

In a first step, we build on a Systematic Mapping Study (SMS) approach (Petersen et al. 2015) to identify characteristics, influence factors, and assessment criteria that have an impact on the readability of test code. In a second step, we complement the mapping study with grey literature to include practical views. Based on identified influence factors, we conducted a controlled experiment (Wohlin et al. 2012) to investigate the perception of readability and understandability in academic environment.

In an initial mapping study (Winkler et al. 2021) we reviewed the scientific literature dedicated to the readability of test code, exploring (a) the demographics of the body of knowledge, (b) the characteristics of the studied test code, and (c) the factors that have shown to impact readability. We analyzed 19 scientific studies filtered from several hundred search results and identified a set of 9 influence factors that have been investigated in academic work either individually or as part of comprehensive readability models. Our mapping study covers the topic of test code readability specifically from the viewpoint of work published in the scientific literature. However, test code readability is of high practical relevance and the topic is therefore also frequently covered in magazine articles, books on testing, and online blogs. These sources are typically referred to as grey literature (Garousi et al. 2019). Based on previous work (Winkler et al. 2021), the goal of this work is to extend the topic

by exploring test code readability in it's entire scope by combining the scientific and the practical viewpoint. We therefore conducted an additional grey literature survey to identify influence factors commonly discussed in practice, we mapped the results to the findings from the previous scientific literature study. Finally, we investigated the newly identified factors in a controlled experiment (Wohlin et al. 2012) comparing the readability of different versions of a selected set of test cases. The main contribution of the paper includes:

1. **Identified influence factors** for the readability of software test code based on a systematic mapping study (SMS) as a combination of academic and practical views, derived from academic and grey literature. Detailed analysis results are available online (Winkler et al. 2023)[1].
2. **Setup and results of a controlled experiment** to investigate influence factors of a selected set of test cases in an academic environment (Winkler et al. 2023).

Consequently, the remainder of this paper is structured as follows: Section 2 describes background and related work on test code quality and code readability. Section 3 defines our research questions, followed by three sections explaining the setup, process and results of the systematic mapping study (Section 4), the grey literature survey (Section 5), and the concluding experiment (Section 6) in context of the respective research questions. Finally, Section 7 summarizes the finding, discusses implications for academia and practitioners and limitations, and identifies future work.

2 Background

The readability of test code is associated to two areas of related research: First, the area of test quality or, more specifically, the quality of code of automated tests (cf. Section 2.1). Second, the related research on source code readability (cf. Section 2.2).

2.1 Software Test Code Quality

In context of software evolution and maintenance, changes made to the software due to bug fixes, extensions, enhancements, and improvements, also require subsequent adaptations of the corresponding tests (Yusifoğlu et al. 2015).Thereby, similar to code quality being an important factor for supporting evolution and maintenance, test code quality is critical for evolving and maintaining tests. Consequently, in test code engineering (Yusifoğlu et al. 2015), the two leading activities are quality assessment and co-maintenance of test-code with production code.

Engineering test code, much like engineering production code, is a challenging process and prone to all kinds of design and coding errors. Hence, test code also contains bugs, which may either cause false alarms (i.e., a test fails although the production code is correct) or which may cause "silent horrors" (i.e., a test passes although the production code is incorrect). Both kind of bugs have been found to be prevalent in practice (Vahabzadeh et al. 2015). The latter kind of bugs are also considered a result of "rotten green tests" (Delplanque et al. 2019), which are tests that pass green but do so by inadequately validating the required properties of the system under test.

Apart from bugs in tests, a widely reported problem related to test code quality are *test smells* (Garousi and Küçük 2018; Spadini et al. 2018; Tufano et al. 2016). Test smells are the

[1] Winkler et al. (2023): https://doi.org/10.48436/w4q8v-28695

equivalent to code smells (Lacerda et al. 2020) (or anti-patterns) in production code, which are symptoms of an underlying problem in the code (e.g., a design problem) that may not cause the software to fail now but bears the risk of causing additional problems and actual bugs in future. Hence, test smells can be considered as poorly-designed tests (similar to rotten green tests) and their presence may negatively affect test suites with respect to the maintainability and even the correctness of the tests (Bavota et al. 2015; Spadini et al. 2018). Although test smells are a popular concept that is frequently investigated in scientific literature, a recent study by Panichella et al. (2022) suggests a mismatch between the definition of test smells and real problems in the tests. To tackle this mismatch they update definitions of test smells and investigate issues which are currently not covered well by the existing smells.

Since the upcoming of test code generators like Evosuite, which aim to generate test suites covering the whole system under test, there is a continuous discussion on improvements and practical usefulness of these tools. For example McMinn et al. (2012) leverage web search engines for generating test data of type `String`. This approach can improve the coverage of the resulting test suites and the use of more realistic input strings could improve the readability of the test code. Various studies (Fraser et al. 2013; Ceccato et al. 2015; Shamshiri et al. 2018) investigate the usefulness of test code generators in debugging activities and also highlight shortcomings of generated test code which relate to the high number of assertions, absence of explanatory comments or documentation, quality of identifiers and in general unrealistic test scenarios. Hence, similar to the quality of automatically generated code (Yetistiren et al. 2022; Al Madi 2022), the quality of generated test code is a critical aspect that requires additional consideration and investigation.

2.2 Readability of Source Code

Reading and understanding source code is an elementary activity in software maintenance and development (Minelli et al. 2015). Hence, code readability has been subject to a wealth of empirical studies; e.g., Oliveira et al. (2020) examining 54 papers on code readability in their literature review. In these studies, a wide range of different factors influencing readability have been investigated, including code formatting and indentation, identifier naming, line length, complexity of expressions, complexity of the control flow, use of code comments, presence of code smells, and many others.

Buse and Weimer (2008) developed a model combining aforementioned factors to automatically estimate code readability. They trained their model on small source code snippets extracted from open-source projects and tagged as readable or non-readable by human annotators. Following this approach, generalized and extended code readability models have been developed in subsequent works, e.g., by Posnett et al. (2011) and Scalabrino et al. (2017).

Buse et al. define *readability* as *"a human judgment of how easy a text is to understand"* (Buse and Weimer 2008). However, no generally accepted definition for the term readability has been established in the literature and, thus, readability is often used in combination with or as synonym for the related terms *legibility* and *understandability*. The term legibility is rather related to the visual appearance of the source code affecting the ability to identify individual elements (Oliveira et al. 2020), while the term understandability is mainly related to semantic aspects of the code. Scalabrino et al. (2017) even developed a model specifically dedicated to the understandability of source code, arguing that program understanding is a non-trivial mental process that requires building high-level abstractions from code to understand its purpose, relationships between code entities, and the latent semantics, which all cannot be sufficiently captured by readability metrics alone.

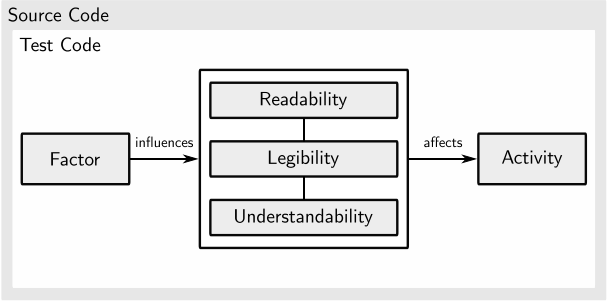


Fig. 1 Concept of readability as used in context of our study

Based on the terms and definitions commonly used in related work, we adopt a broader view on the concept of readability embracing all three terms – *readability*, *legibility* and *understandability* – in our work in context of test code readability. Hence, in the remainder of this paper, we use the term readability subsuming all related notations since it is the most commonly used term in the software engineering literature.

Figure 1 depicts this view and the distinction between factors (e.g., test case length) influencing readability as well as upstream activities (e.g., test case maintenance) being affected by readability. The underlying concept is related to activity-based quality modeling as proposed in Quamoco approach (Wagner et al. 2015, 2012) and used, e.g., for modeling maintainability (Deissenboeck et al. 2007) or requirements quality (Femmer et al. 2015). Readability is described by more fine-grained factors that can be assessed (manually or automatically) and it has an observable impact on the activities performed by stakeholders related to a specific entity. In our context, the entity of interest is the test code, shown as a subset of source code.

3 Research Questions

Based on the goal of this article to investigate the readability of software test code by combining scientific and practical views, we identified three groups of research questions, with focus on (a) influence factors in academia, (b) influence factors in practice, and (c) an investigation of combined influence factors on a selected set of test cases in a controlled experiment.

3.1 Influence Factors in Academia

Based on our previous work, an initial Systematic Mapping Study (SMS) Winkler et al. (2021), we extended the mapping study by introducing additional analysis criteria. Therefore, we identified the first research question (RQ1) and two sub-research questions to (a) identify influence factors (RQ1.1) and (b) to explore methods (RQ1.2) used in scientific studies. We applied the Systematic Mapping Study (SMS) approach, proposed by Petersen et al. (2015). Section 4 describes the research protocol and the results of the mapping study.

RQ1. Influence factors on test code readability in scientific literature?
RQ1.1 Which influence factors are analyzed in scientific literature?
RQ1.2 Which methods are used in scientific studies?

3.2 Influence Factors in Practice

To systematically capture influence factors, discussed in industry and practice, we extended and complemented the mapping studies with grey literature. The results will show similarities and differences of academia and industry in context of the readability of software test code. For investigating grey literature, we followed the guidelines proposed by Garousi et al. (2019). Section 5 presents the research protocol and the results.

RQ2. Influence factors on test code readability discussed in practice?
RQ2.1 Which influence factors are discussed in grey literature?
RQ2.2 What is the difference between influence factors in scientific literature and grey literature?

3.3 Investigating Influence Factors in a Controlled Experiment

Based on identified factors, we conducted a controlled experiment in academic environment to investigate the perception of readability and understandability of a selected set of test cases (derived from open source projects). We build on the guidelines, proposed by Wohlin et al. (2012) for planning, executing, and reporting on the controlled experiment. Section 6 presents the experiment setup and reports on the results.

RQ3. What is the influence on the test code readability of discussed factors?
RQ3.1 Do factors discussed in practice show an influence on readability when scientific methods are used?

4 Systematic Mapping Study

To investigate the *Influence Factors on Readability in Scientific Literature*, we conducted a *Systematic Mapping Study (SMS)* based on Petersen et al. (2015).

In this section, we summarize the study protocol and the results from the systematic mapping study (SMS) (Winkler et al. 2023). We present influence factors and study types with focus on research methods used.

4.1 Study Protocol and Process

This section summarizes the study protocol with focus on the systematic mapping study of scientific publications.

An integral part of systematic mapping studies as proposed by Petersen et al. (2015) is the thorough documentation of the process to make the results traceable. Figure 2 provides an overview of the whole process. After the initial search (Step 1) and filtering (Step 2), we apply back- and forward snowballing (Step 3) and filter (Step 4) to obtain our final set of studies. We repeat the steps 3 and 4, the back- and forward snowballing, until exhaustion, i.e., they add no new relevant studies to the result set. In our case, no additional publications were identified in the second iteration. The following subsections provide details on each of these steps.

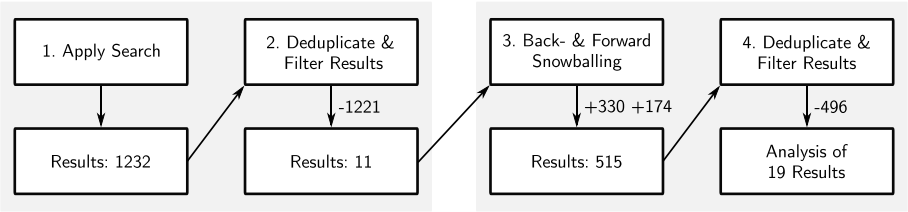


Fig. 2 Systematic mapping study process and amount of received publications

Step 1: Apply Search Based on the research questions (cf. Section 3, RQ1), we defined the following *keywords*: *test, code, model, readability, understandability, legibility* and *smell*. We used them to build the *Search Strings* shown in Table 1. The queries were performed on established sources for scientific literature, i.e., *Scopus*, *IEEE* and *ACM*, and we filtered the studies based on title, abstract and keywords. In the ACM search we enclosed the term "understandability" in double quotes in the abstract filter to enforce exact matching, because ACM's fuzzy matching leads to a high number of irrelevant results. For ACM we searched in the *ACM Guide to Computing Literature* which offers a larger search space than the *ACM Full-Text Collection*. The search was conducted at the end of November 2021 without limiting the publication year and returned a total of 1232 raw results (Scopus: 460, IEEE: 231, ACM:541). Based on the merged results, we proceeded to the next step.

Step 2: Deduplicate & Filter Results We first deduplicated the raw results based on the digital object identifier (DOI) and title, which removed 281 studies. Next, we imported the result set into a spreadsheet solution for applying inclusion and exclusion criteria.

Inclusion Criteria. We included a study if both of the following criteria were fulfilled:

- Conference papers, journal/magazine articles, or PhD theses (returned by ACM)
- Readability, understandability or legibility of test code is an object of the study

Exclusion Criteria. We excluded a study if one of the following criteria is applicable:

Table 1 Search strings in different databases

Database	Search string
Scopus	SUBJAREA (COMP) TITLE-ABS-KEY(((code) AND (test* OR model) AND (readability OR understandability OR legibility)) OR ((“test” OR “code”) AND (smell) AND (readab* OR understandab* OR legib*)))
IEEE	((“All Metadata”: code) AND (“All Metadata”: test* OR “All Metadata”: model) AND (“All Metadata”: readability OR “All Metadata”: understandability OR “All Metadata”: legibility)) ((“All Metadata”: “test” OR “All Metadata”: “code”) AND (“All Metadata”: smell) AND (“All Metadata”: readab* OR “All Metadata”: understandab* OR “All Metadata”: legib*))
ACM	((Title:(code) AND Title:(test* model) AND Title:(readability understandability legibility)) OR (Keyword:(code) AND Keyword:(test* model) AND Keyword:(readability understandability legibility)) OR (Abstract:(code) AND Abstract:(test* model) AND Abstract:(readability “understandability” legibility))) OR ((Abstract:(“test” “code”) AND Abstract:(smell) AND Abstract:(readab* understandab* legib*)) OR (Keyword:(“test” “code”) AND Keyword:(smell) AND Keyword:(readab* understandab* legib*)) OR (Title:(“test” “code”) AND Title:(smell) AND Title:(readab* understandab* legib*)))

- Not written in English
- Conference summaries, talks, books, master thesis
- Duplicate or superseded studies
- Studies not identifying factors that influence test code readability

The criteria were evaluated based on title and abstract of the results by at least one of the authors. When in doubt about including or excluding, the evaluated study was discussed with a second evaluator. This step left us with 11 scientific publications.

Backward & Forward Snowballing - First Iteration Based on the initial iteration, we executed backward & forward snowballing to identify relevant studies that have not been identified in the initial search.

- **Step 3: Backward & Forward Snowballing.** Since relevant literature might refer to further important studies, we used the references included in the 11 studies for backward snowballing via Scopus. The 11 studies might also be cited by other relevant studies, hence we also performed forward snowballing, by using Scopus to find studies, which cite one of the initial 11 studies. This increased the result set by 330 from backward snowballing and 174 from forward snowballing to a total of 515 studies.
- **Step 4: Deduplicate & Filter Results.** By comparing these 515 studies with the initial result set we found and removed 83 duplicates. Similar to step 2, one of the authors of this paper applied the inclusion and exclusion criteria. Additionally, after a full text reading, all studies were discussed and reevaluated by the author team. With this, we reduced the result set by 496 and obtained a final result of 19 studies.

Backward & Forward Snowballing - Second Iteration We performed a second iteration of back- and forward snowballing via Scopus with these 19 studies as input. This returned a raw result of 825, which we reduced to 357 studies by removing duplicates. We applied in- and exclusion criteria on the remaining studies, which removed all 357 studies. Therefore, this second iteration did not add any new relevant studies to the results set of 19 studies.

Studies Not Included In the following, we provide four exemplary cases filtered out in step 2 and the rationale why these studies did not meet the criteria for inclusion in the final publication set after discussion by all authors: Grano et al. (2020) focus on semi-structured interviews with five developers from industry and a confirmatory online survey to synthesize which factors matter for test code quality. Although readability is seen as a critical factor by all participants, the analysis of readability and influencing factors was not in the scope of this work. Tran et al. (2019) investigated general factors for test quality by interviewing 6 developers from a company. Quality factors are discussed with natural language tests brought by the participants. Since our work has its specific focus on test code, readability of natural language tests was not considered further. Bavota et al. (2015) report on four lab experiments with an overall number of 49 students and 12 practitioners and effects on maintenance tasks from eight test smells. These test smells occur frequently in software systems. While this work clearly shows that test smells negatively affect correctness and effort for specific maintenance tasks, a connection between test smells and readability is not shown. Deiß (2008) reports on a case study about semi-automatic conversion and refactoring of a TTCN-2 test suite to TTCN-3. Discussed improvements included reducing complicated or unnecessary code artifacts generated by the automatic conversions that are also supposed to increase readability. We excluded this study since the focus was the migration from TTCN-2 to TTCN-3 and the study did not investigate the improvements on the readability of test code.

Table 2 Final set of publications based on the search process

Idx	Title	Authors	Venue	Year	Study Type
[*A*17]	Developer's Perspectives on Unit Test Cases Understandability	Setiani N. et al.	ICSESS	2021	Experiment + Survey (hum)
[*A*16]	DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests	Roy D. et al.	ASE	2020	Experiment + Survey (hum)
[*A*18]	Test case understandability model	Setiani N. et al.	IEEE Access	2020	Experiment (hum)
[*A*13]	On the quality of identifiers in test code	Lin B. et al.	SCAM	2019	Survey (hum)
[*A*3]	What Factors Make SQL Test Cases Understandable for Testers? A Human Study of Automated Test Data Generation Techniques	Alsharif A. et al.	ICSME	2019	Experiment + Survey (hum)
[*A*10]	Fluent vs basic assertions in Java: An empirical study	Leotta M. et al.	QUATIC	2018	Experiment (hum)
[*A*9]	An empirical investigation on the readability of manual and generated test cases	Grano G. et al.	ICPC	2018	Experiment
[*A*12]	Aiding comprehension of unit test cases and test suites with stereotype-based tagging	Li B. et al.	ICPC	2018	Experiment + User Study (hum)
[*A*7]	Specification-Based testing in software engineering courses	Fisher G. and Johnson C.	SIGCSE	2018	Experiment + Survey (hum)
[*A*2]	An industrial evaluation of unit test generation: Finding real faults in a financial application	Almasi M. et al.	ICSE-SEIP	2017	Experiment + Survey (hum)
[*A*6]	Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?	Daka E. et al.	ISSTA	2017	Experiment + Survey (hum)
[*A*4]	How Good Are My Tests?	Bowes D. et al.	WETSoM	2017	Concept paper (hum)
[*A*14]	Automatic test case generation: What if test code quality matters?	Palomba F. et al.	ISSTA	2016	Experiment
[*A*11]	Automatically Documenting Unit Test Cases	Li B. et al.	ICST	2016	User study (hum)

Table 2 continued

Idx	Title	Authors	Venue	Year	Study Type
[*A*15]	The impact of test case summaries on bug fixing performance: An empirical investigation	Panichella S. et al.	ICSE	2016	Experiment (hum)
[*A*19]	Towards automatically generating descriptive names for unit tests	Zhang B. et al.	ASE	2016	Prototype and User Study (hum)
[*A*5]	Modeling readability to improve unit tests	Daka E. et al.	ESEC/FSE	2015	Experiment + Survey (hum)
[*A*1]	Evolving readable string test inputs using a natural language model to reduce human oracle cost	Afshan S. et al.	ICST	2013	Experiment (hum)
[*A*8]	Exploiting common object usage in test case generation	Fraser G. and Zeller A.	ICST	2011	Experiment

4.2 Systematic Mapping Study Results

This section summarizes the findings in context of influence factors on readability found in scientific literature (Table 2).

4.2.1 Which influence factors are analyzed in scientific literature (RQ1.1)?

In RQ1, we explore the factors that have been found to impact readability of test code.

Influence factors have been derived by one of the authors based on the content of the paper. Other authors and testing experts have reviewed the initially identified factors. Deviations have been discussed by all authors to come to a consensus. Table 3 maps candidate factors to the studies that investigate them. Two approaches of how influence factors are considered in the primary studies can be distinguished. Studies either **(a)** investigate the impact of one or more *individual factors*, often related to the attempt to improve readability with a specific approach or tool, or **(b)** they target *readability models* constructed from a combination of many factors. The majority of the primary studies (see Table 3a) consider individual factors. Readability models were subject to study only in three instances (Table 3b), although such models are commonly used in the general research on source code readability.

We identified a total of 9 unique influence factors in the scientific literature as shown in Tables 3a and b. In the following, we briefly explain these factors. The number in brackets shows the number of primary studies including the particular factor combining counts from both tables.

Test names (6) The name of the test method or test case. Not only generated tests have poor names but also names provided by humans often convey few useful information. Therefore, several studies propose different solutions on automatic test renaming e.g., Zang et al. [A19] Roy et al. [A16] or Daka et al. [A6]. In studies from e.g. Setiani et al. [A17], Bowes et al. [A4] or Panichella et al. [A15] participants agree on the importance of test names for readability.

Table 3 Reported factors influencing test code readability

(a) Studies investigating individual factors																			
Individual factors	[A17]	[A16]	[A18]	[A13]	[A3]	[A10]	[A9]	[A12]	[A7]	[A2]	[A6]	[A4]	[A14]	[A11]	[A15]	[A19]	[A5]	[A1]	[A8]
Test names (6)	•	•									•	•			•	•			
Assertions (5)	•		•			•				•							•		
Identifier names (5)	•	•		•					•			•							
Test data (4)					•					•		•						•	
Test summaries (4)		•						•						•	•				
Dependencies (3)	•												•						•
Comments (2)	•								•										
Test structure (2)	•											•							
(b) Studies using readability models																			
Readab. models	[A17]	[A16]	[A18]	[A13]	[A3]	[A10]	[A9]	[A12]	[A7]	[A2]	[A6]	[A4]	[A14]	[A11]	[A15]	[A19]	[A5]	[A1]	[A8]
Test structure (3)			•				•										•		
Textual features(1)							•												

Assertions (5) This factor relates to the amount of assertions in a test case as well as to assertion messages. Although assertions are an integral part of test code Daka et al. [A5] report minor influence on readability coming from the amount of assertions in a test. Nevertheless the amount of assertions is still used in their readability model and also in the model from Setiani et al. [A18]. In the survey from Setiani et al. [A17] assertions are mentioned to have an influence, but other factors like naming are deemed more important. In Almasi et al. [A2] developers issued concerns on generated assertions. Leotta et al. [A10] find no significant effects on readability when AssertJ assertions are used instead of basic JUnit assertions, although other positive effects could be observed.

Identifier names (5) Naming of variable names in test cases. Especially Lin et al. [A13] investigate this factor thoroughly and also provide characteristics of good and bad identifier names based on a survey. Roy et al. [A16] propose an automatic way for identifier renaming in test cases. In studies from e.g. Setiani et al. [A17] and Bowes et al. [A4] participants agree on the importance of identifier names for readability. Fisher and Johnson [A7] attribute differences between generated and manually written tests to differences in naming.

Test structure (2+3) Structural features are found in studies investigating individual factors (2 times) as well as in readability models (3 times). They include strucural features of test methods like maximum line length, number of identifiers, length of identifiers, number of control structures (e.g., branching as mentioned in Bowes et al. [A4], test length, etc. Participants in the study from Setiani et al. [A17] agree on the importance of the amount of lines of code in the tests. These features are also used in combination by automatic readability raters, e.g., from Daka et al. [A9], who propose a rater especially for test cases, Grano et al. [A9] or Setiani et al. [A18].

Test data (4) Testers often have to evaluate data used in assertions to decide if a test has truly failed or if there is a fault in the test. Afshan et al. [A1] investigates this topic and shows that readable string test data helps humans predicting correct outcome. Alsharif et al. [A3] and Almasi et al. [A2] highlight the importance of meaningful test data. Furthermore, in the workshop study from Bowes et al. [A4] developers, amongst others, state that *magic numbers* are detrimental to readability.

Test summaries (4) Documentation describing the whole test case support understanding what the tests do, for example as Javadoc like in Roy et al. [A16] or Li et al. [A11] or interleaved with test code like in Panichella et al. [A15]. Li et al. [A12] reduce the amount of generated description, by only adding test stereotypes as tags.

Dependencies (3) The number of classes a single test case depends on, as proposed by Fraser et al. [A8], or if a test is truly a unit test and therefore independent from other parts of the system as reported by Setiani et al. [A17]. Test coupling and cohesion discussed by Palomba et al. [A14] describe dependencies between tests and are included in this factor.

Comments (2) Single comments in test code providing useful information. According to Fisher and Johnson [A7], one of the differences between their generated tests and human tests is the lack of explanatory comments. In Setiani et al. [A17], survey participants also mention comments being to some degree important to readability.

Textual features (1) Textual features focus on natural language properties part of test cases like consistency of identifiers or identifiers present in a dictionary. These features can be easily computed and are therefore frequently used in readability models and automatic readability raters like in Scalabrino et al. (2016).

RQ1.1 Findings. Which influence factors are analyzed in scientific literature? A total of 9 influence factors have been found in the scientific literature, which can be grouped into individual factors and readability models. The three most frequently mentioned individual factors are *test names*, *assertions*, and *identifier names*. Readability models combine several individual factors related to *test structure* or *textual features*.

4.2.2 Which research methods are used in scientific studies (RQ1.2)?

In RQ1.2, we give an overview on the study types and the used methods. This analysis is based on the classification of established empirical research methods involving human participants Wohlin et al. (2012). Although software tools for investigating the readability of software code exist, the readability of software tests is not in the main focus of these approaches.

Concerning the utilized types shown in Table 2, most studies (15) report an experiment which is combined with a survey in 7 studies. Human involvement is quite common, in 16 from 19 studies humans take part in experiments, surveys or play another role as participants of the study. Next, we present details on the individual types of studies.

Experiment (15) 12 of 15 studies evaluate the effect of an approach with humans by either asking participants to answer questions to a given test case or code snippet without knowing the origin like in Roy et al. [A16] or Daka et al. [A6] or participants have to choose between two versions (forced choice) like in Setiani et al. [A17] or Daka et al. [A5]. Alsharif et al. [A3] enhance their experiment by letting some participants vocalize their thoughts while filling out a questionnaire in a Think Aloud Study. Li et al. [A12] do not fit in this categorisation. They use an indirect approach to measure the effect of generated tags by letting one group write summaries of test cases with and without treatment. Another group then rates these summaries according to a scheme. The difference in the ratings shows the effect of the treatment. For analysing the experiments results, eight from 15 studies use a form of the Wilcoxon test, most commonly the Wilcoxon rank sum test. Furthermore, these studies report the effect size with the Vargha-Delaney ($\hat{A}_{12}$) statistic or Cliff's Delta. Three of these studies also use the Shapiro-Wilk test for normal distribution to decide if a parameterized test can be applied. Alsharif et al. [A3] use a Fisher's Exact test on their results. The remaining studies interpret the results without statistical tests.

Survey (8) Five out of eight studies use online questionnaires, one uses an off-site questionnaire and for one study the kind of survey could not be extracted. In the surveys six out of eight studies use Likert scales often for rating readability. Free text answers are also common for optionally elaborating on a rating or as a mitigation against random readability ratings like in Daka et al. [A6].

User study and Prototype (3) The three studies of these types use surveys with Likert scale in Lin et al. [A13], forced choice questions with opportunity to elaborate on the rating in Zhang et al. [A19] or a mixture of multiple- and binary-choice and open questions in Li et

al. [A12]. Zhang et al. [A19] use the Wilcoxon test for comparing the results of a prototype tool with other tools after a test on normality with the Shapiro-Wilk test.

Concept Paper (1) Bowes et al. [A4] brainstorm and discuss quality evaluation of software tests with industry partners. Afterwards they merge the result with their own teaching experience and relevant scientific literature and books on software testing.

RQ1.2 Findings. Which research methods are used in scientific studies? For gathering humans opinion on readability online questionnaires with Likert scales and free text answers are common. The dominant result analysis consists of a statistical analysis with a Wilcoxon test after an optional test on normality with the Shapiro-Wilk test.

5 Grey Literature Review

In this section, we first describe the study protocol and process for the grey literature analysis followed by presentation of the results including a discussion of the respective research questions. The data set is available online (Winkler et al. 2023).

5.1 Study Protocol and Process

The process for conducting the review of grey literature (Fig. 3) is similar to the scientific literature review, except that there is no backward snowballing. The guidelines and recommendations by Garousi et al. (2019) were used as input for this part of our work. We decided to add grey literature to this work, because testing is frequently performed by practitioners, and we assume that for them the internet is one of the first places used for information gathering and sharing.

Step 1: Apply Search Based on the research questions and knowledge obtained from the previous literature search we used the search strings *"test code" readability* and *"test code" understandability*. We performed these queries separately on *Google* using a script for extracting all results. The script mimics a search without being logged in with a Google account. Therefore personalized search results should be reduced to some degree. In contrast to Googles prediction of hundreds of thousands of results, it returned 146 results for *"test*

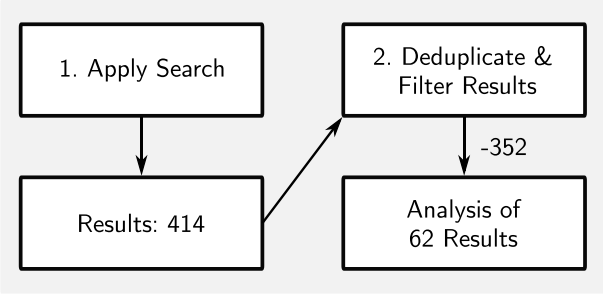


Fig. 3 Grey literature review process and amount of received grey sources

code" readability, 101 for *"test code" understandability* and 167 for *"test code" legibility* (total: 414) in mid-February 2022.

Step 2: Deduplicate & Filter Results We first deduplicate the results by comparing the links which removed 18 sources. The result set was imported into a spreadsheet for applying inclusion and exclusion criteria.

Inclusion Criteria. We included a study if the following criterion was fulfilled:

- Readability or understandability of test code is a relevant part of the source. This is the case if the length of the content on readability is sufficient and if the source contains concrete examples of factors influencing readability.

Exclusion Criteria. We excluded a study if one of following criteria applied:

- Not written in English
- Literature indexed by *ACM*, *Scopus*, *IEEE*
- Duplicates, videos, dead links

The criteria were evaluated based on the contents of the source. This step left us with 62 results ready for further analysis and extraction of influence factors.

Excluded Sources. Similar to the scientific literature search, we provide some examples and rationale for sources that where excluded when applying the defined criteria: Source Karhik [2] is a blog entry, which is relatively short and primarily lists features of AssertJ. Although the entry mentions readability improvement by using AssertJ in one sentence, it gives no explanation for this claim. Source Karlo Smid[3] discusses the DRY-principle (don't repeat yourself) in context of unit testing. However, the blog entry is very short and primarily references to another source already present in the result set [G59]. Although the collaborative source Openstack[4] has a reasonable size and it also has a section on readability, the statements are too generic and do not contain a concrete influence factor on readability. Finally there are also many sources which are off-topic, e.g, because they discuss general code readability or quality, they describe advantages of unit testing, or they are documentation pages of test frameworks.

5.2 Grey Literature Analysis Results

In the following, we present the results from our further analysis of the grey literature sources with regard to factors influencing readability, and we provide answers to the research questions RQ2.1 and RQ2.2.

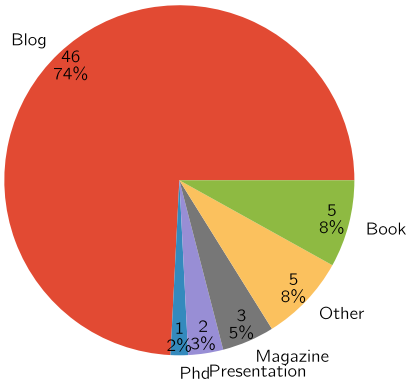
Infuence factors. We identified 12 types of influence factors in the analysis of the gray literature. The factors are related to *test structures (Str)*, *test names (TeN)*, *assertions (Asse)*, *helper structures (Help)*, *dependencies (Dep)*, *identifier names (IdN)*, *fixtures (Fix)*, *DRY principle (DRY)*, *test data (TeD)*, *comments (Com)*, *domain specific language (DSL)*, and *parameterized test (Par)*. A detailed description of each factor is provided in Section 5.2.1.

The influence factors were extracted from the literature sources by one of the authors by tagging each source with keywords, which are mentioned in the context of test code

[2] Karhik, Use AssertJ to improve your test code readability ... - Upnxtblog, https://www.upnxtblog.com/index.php/2018/04/25/use-assertj-to-improve-your-test-code-readability-maintenance-of-tests-easier/

[3] Karlo Smid, Kill The Unit Test - Tentamen Software Testing Blog, https://blog.tentamen.eu/kill-the-unit-test/

[4] TestGuide - OpenStack wiki, https://wiki.openstack.org/wiki/TestGuide



Type	Absolute	Relative
Blog	46	74%
Book	5	8%
Other	5	8%
Magazine	3	5%
Presentation	2	3%
Phd	1	2%
Sum:	62	100%

Fig. 4 Types of analyzed selected grey sources

readability. Keyword mentioned in a different context were not included. For example, in [G49] the use of “helper methods" is only mentioned in context of easier maintenance, therefore this appearance of the factor helper methods is not counted. The results were cross-checked and discussed by the other authors of the study.

In our analysis, we also investigated what types of gray literature sources we analyzed, when the literature sources mentioning the influence factors were published, and in context of what programming languages readability was discussed.

Source types. Figure 4 shows the identified types of grey literature source. From the 62 sources around 75% (46 in total) are identified as blog entries of various sizes. A source is also identified as a blog when there is no clear indication that an editorial team is involved. The types of the remaining 16 sources are spread across 5 books, 5 other types (stackoverflow, quora, wiki, cheatsheet, podcast), 3 magazines, 2 presentations (slide shows), and 1 Phd thesis.

Factor across years. Figure 5 shows the factors investigated by the blogs across the years. The bottom line *Sources per year* gives the number of sources in a particular year which investigated the factors above. Apart from parameterized tests, which appeared only seven times in the years 2020 and 2021 and in fewer sources in 2017 and 2018, there are no obvious fluctuations in the distribution of factors. Table 4 shows the selected sources ordered by years descending and the investigated factors in detail, where these effects are also visible.

Programming languages. Concerning programming languages, 19 sources mention Java or use Java code snippets, C# appears in 10, Java Script in nine sources and, Ruby in three sources. Kotlin and Python each appear in two sources, Scala, Typescript, C++ and Go are mentioned in one source each. Some sources do not mention a certain programming language or do not use code snippets, because they provide general best practices for testing. This is in accordance with the findings of our previous SMS (Winkler et al. 2021), where Java is the dominant language used in studies on test code readability.

5.2.1 Which influence factors are discussed in grey literature (RQ2.1)?

In the following, each of the 12 influence factors identified in the gray literature analysis is described in detail. The number in brackets shows how many of the 62 reviewed literature sources mention the factor. They range from 28 (45%) to 8 times (13%). Table 4 lists the analyzed grey literature sources (rows) and shows in which of these sources the identified

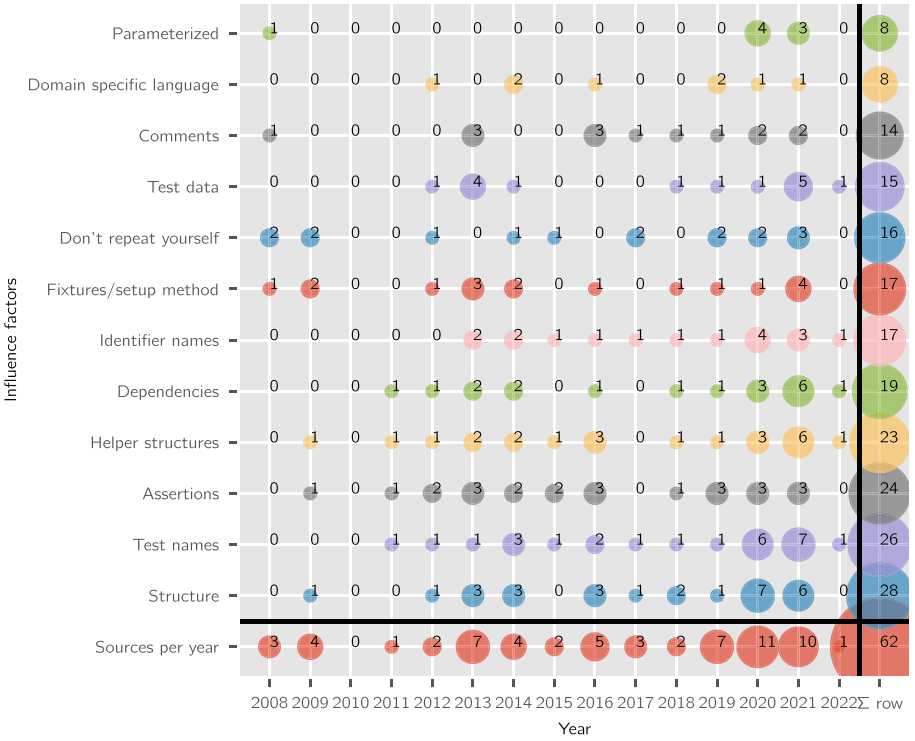


Fig. 5 Factors investigated by grey literature. The bottom row gives the total number of sources per year, which may cover multiple factors

influence factors (columns) are mentioned. For the sake of completeness, the table shows all 14 influence factors identified in the scientific as well as in the gray literature search, which includes two factors not mentioned in the gray literature.

Test structure (28) (Str): 23 out of 28 sources suggest the use of patterns like *Arrange, Act, Assert* [G31], *Given, When, Then* [G18] or *Build, Operate, Check* [G54]. Two sources ([G35] and [G25]) suggest to group similar test cases to see differences more quickly. Other sources [G52][G22] suggest to watch out for "eye-jumps", e.g., a variable, which is initialized many line breaks away from its usage. The absence of logic, shortness, and coherent formatting of test cases is also mentioned by several authors.

Test names (26) (TeN): All sources suggest coherent naming of test cases and most of them suggest a concrete naming pattern like *givenFooWhenBarThenBaz* [G3] or *subject_scenario_outcome* [G57]. Three sources ([G31], [G13] and [G28]), explicitly suggest to use spaces in test names, which is a practice also shown by others in code examples, e.g., [G57] and [G5]. Long names are explicitly okay for two sources, since these methods are not called in other parts of the code. Different opinions exist on the inclusion of the name of the concrete tested method in the test name. [G20], [G34] and [G52] suggest to include the method name in the test name. Other sources like [G41], [G11] and [G9] do not recommend to include the method name, because if the method name changes the test name has to change too. Instead the tested behavior should be described.

Table 4 Factors influencing readability mapped to sources from the grey literature search

	Str	TeN	Asse	Help	Dep	IdN	Fix	DRY	TeD	Com	DSL	Par	TS TF
[*G*28]		•		•	•	•	•	•	•			•	
[*G*15]	•	•	•			•	•	•	•	•			
[*G*61]	•			•	•		•		•	•		•	
[*G*31]	•	•	•	•	•	•				•			
[*G*3]		•	•	•	•	•	•		•				
[*G*52]	•	•		•		•	•		•	•			
[*G*34]		•	•	•	•	•			•		•		
[*G*43]	•		•	•	•		•				•		
[*G*5]	•	•			•	•			•			•	
[*G*25]	•		•	•	•		•			•			
[*G*21]	•	•			•	•	•				•		
[*G*1]		•		•	•	•			•				
[*G*47]	•	•		•	•	•							
[*G*18]	•		•	•						•	•		
[*G*11]		•	•	•		•		•					
[*G*62]	•	•	•	•			•						
[*G*20]		•	•	•					•		•		
[*G*35]	•		•		•		•	•					
[*G*4]		•						•	•			•	
[*G*56]	•	•			•				•				
[*G*57]	•	•			•					•			
[*G*22]	•						•				•	•	
[*G*8]	•	•	•			•							
[*G*27]	•		•	•						•			
[*G*19]	•		•	•						•			
[*G*37]			•		•		•		•				
[*G*17]	•			•			•			•			
[*G*10]			•		•		•		•				
[*G*9]		•	•	•	•								
[*G*26]	•	•										•	
[*G*54]	•		•		•								
[*G*46]	•	•				•							
[*G*41]	•	•								•			
[*G*40]				•		•			•				
[*G*60]	•					•			•				
[*G*45]	•			•				•					
[*G*44]	•	•											
[*G*39]	•	•											
[*G*53]				•				•					
[*G*58]			•	•									

Table 4 continued

	Str	TeN	Asse	Help	Dep	IdN	Fix	DRY	TeD	Com	DSL	Par	TS	TF
[*G*30]								•				•		
[*G*12]		•				•								
[*G*2]			•	•										
[*G*24]								•		•				
[*G*49]		•				•								
[*G*7]	•							•						
[*G*6]			•							•				
[*G*16]							•	•						
[*G*36]								•		•				
[*G*23]							•					•		
[*G*29]								•						
[*G*55]					•									
[*G*50]			•											
[*G*42]											•			
[*G*59]								•						
[*G*38]											•			
[*G*14]								•						
[*G*13]		•												
[*G*51]			•											
[*G*32]			•											
[*G*48]							•							
[*G*33]								•						
Total	28	26	24	23	19	17	17	16	15	14	8	8	–	–

New factors identified in the gray literature are shown in purple. **Str**: structure, **TeN**: test names, **Asse**: assertions, **Help**: helper structures, **Dep**: dependencies, **IdN**: identifier names, **Fix**: fixtures, **DRY**: DRY principle, **TeD**: test data, **Com**: comments, **DSL**: domain specific language, **Par**: parameterized test, TS: test summaries, TF: textual features

Assertions (24) (Asse): The use of appropriate assertions or custom assertions is suggested in eleven sources, e.g., [G9] and [G3]. Nine sources mention assertion libraries like AssertJ (Java) or FluentAssertions (C#) since they enable a more natural language style for asserting properties and contain additional assertions for collection types [G50][G18]. Four sources stress the importance of assertion messages for debugging. Concerning the amount of assertions, the rule "one assertion per test" is mentioned by, e.g., [G31] and [G25].

Helper structures (23) (Help): 13 sources recommend helper methods in order to hide (irrelevant) details like creating objects or asserting properties [G27][G19]. The *Builder Pattern* (or similar patterns) are used by six sources for creating the objects under test, e.g., [G45][G61]. Inheritance of test classes is seen critically by some authors, e.g., [G53][G62].

Dependencies (19) (Dep): All 19 sources agree that one test should only test one functionality or behavior. This affects readability positively, because the test stays short and the test name can be more descriptive, since only one behavior has to be described. Four sources high-

light to only assert properties which are absolutely necessary for the functionality described by the test name and to resist the urge to check additional properties.

Identifier names (17) (IdN): While nine sources only give generic information (e.g. *should have meaningful or intention revealing names*), other sources provide detailed recommendations suggesting, e.g., to either prefix variables with *expected* and *actual* [G28] or use names like *testee, expected, actual* [G11].

Fixtures (17) (Fix): Although 15 of the 17 sources use fixtures, sometimes in combination with setup methods, two sources [G62][G52] argue against the use of fixtures, because they are not visible in the test itself and may contain important information. Similarly, [G28] points out that moving reusable test data into a fixture forces the reader to jump between two locations. Finally, [G21] suggests that fixtures should only be used for infrastructure and not for the system under test, and [G15] recommends to use them only for properties which are needed in every test case.

DRY principle (16) (DRY): In the sources which mention the ***D**on't **R**epeat **Y**ourself* (DRY) principle, there is an agreement that strict adherence to this principle hides away information important for understanding test cases. Others favour the ***D**escriptive **A**nd **M**eaningful **P**hrases* (DAMP) principle [G11] or to find a balance between these principles. As a combination of both, two sources [G36][G53] suggest to clearly show what a test does (DAMP), but to hide how it is done in a helper method (DRY).

Test data (15) (TeD): Five authors suggest to avoid literal test data (a.k.a. magic values), instead local variables, constants or helper functions should be used to provide additional information, e.g., [G37][G34]. However, [G28] argues that declaring local variables for this purpose can quickly increase the test size and the mapping between variable and actual value has to be kept in mind when reading the test. Similarly, [G15] states that using literal values instead of variables sometimes improves readability. Finally, test data should be production-like and simple, and one author also recommends to highlight important data.

Comments (14) (Com): Eleven sources use comments in their snippets or mention them in the text to highlight *Arrange, Act, Assert* or similar structures. However, this is not a strict rule for every author, e.g., source [G57] uses empty lines as an alternative or [G18] mentions to use comments with respect to the capabilities of the testing framework. If the framework already provides such structural hints then comments are unnecessary. Common code comments are mentioned by three sources with the general advice to avoid them, e.g., [G25].

Domain specific language (8) (DSL): In order to make tests more readable also for non programmers, some sources, e.g., [G38][G42], suggest using helper functions or Gherkin (applied in Behavior Driven Testing with Cucumber) as domain specific languages. Such languages describe the executed behavior in natural language and, thus, hide the execution details.

Parameterized test (8) (Par): Eight sources suggest to use parameterized tests (aka data-driven or table-driven tests) to reduce code duplication. This is also suggested by authors who are not in strict favor of the DRY principle, e.g., [G28].

RQ2.1 Findings. Which influence factors are discussed in grey literature? A total of 12 influence factors were found in the gray literature. They were mentioned from 28 to 8 times in the 62 analyzed sources. The five most often mentioned factors are related to test structures (28), test names (26), assertions (24), helper methods (23), and dependencies (19).

5.2.2 What is the difference between influence factors in scientific literature and grey literature (RQ2.2)?

From the total 12 factors identified in the grey literature review, 7 were already known from the scientific literature, while 5 factors were only found in the grey literature. These factors are new and did not appear in our previous white literature study Winkler et al. (2021). In the scientific literature review we identified a total of 9 factors. It included two factors, which were mentioned only in the scientific literature but not in the gray literature. In the description below, the numbers in the brackets (*A* vs *B*) indicate how often a factor was found in the scientific literature versus in the grey literature.

However, even if factors have been found in both sources, the specific view on a factor can sometimes vary between white and gray literature. For example, quantifiable structural properties like line length or number of identifiers tend to be in the focus of scientific literature, whereas grey literature sources focus more on the semantic structure, e.g., the *Arrange-Act-Assert* pattern. Table 5 provides an overview of the differences identified in our analysis.

Test structure (5 vs 28) (Str): Literature published in academic context tends to focus more on countable properties like maximum line length, amount of control structures, etc. (see, e.g., Grano et al. [A9], Daka et al. [A5] or Setiani et al. [A18]). In contrast, the authors of grey literature sources focus on a semantic form of structure like the AAA pattern, which is also discussed in another study by Setiani et al. [A17]. They report moderate positive influence on readability from this *Arrange, Act, Assert* structure.

Test names (6 vs 26) (TeN): Like in the grey literature, scientific literature also mentiones the use of naming patterns, e.g., when test cases are renamed. Zhang et al. [A19] or Daka et al. [A6] use *testSubjectOutcomeScenario* where "Subject" is the method under test, although *outcome* and *scenario* can be left out. The approach by Roy et al. [A16] generates test names with a machine learning model based on the body of the test. According to examples given in the paper, this approach does not seem that it has to include the concrete method under test in the name. In other studies, e.g., Panichella et al. [A15] or Setiani et al. [A18], survey participants highlight the importance of meaningful test names.

Assertions (5 vs 24) (Asse): Some grey sources suggest to apply the "one assertion per test" rule. However, there is little evidence in scientific literature about the effect of assertions on readability. Setiani et al. [A17] report low influence from assertion messages on readability. Furthermore, Setiani et al. [A17] and Daka et al. report negligible influence from the amount of assertions. Studies like Bai et al. (2022) or Panichella et al. (2022) from the field of test smells confirm the negligible importance of assertion messages and the number of assertions. Almasi et al. [A2] report concerns from developers about the meaningfulness of generated assertions. Leotta et al. [A10] report no significant influence on test comprehension when AssertJ is used instead of basic JUnit assertions. This contradicts the voices from grey literature, which suggest to improve readability with fluent assertions.

Table 5 Differences in influence factors between scientific and grey literature

Scientific Literature	Grey Literature
Test structure (5 vs 28)	
Identifier length, line length	
Constructor calls	
Number of identifiers	
Control structure	Control structure
Length of test case	Length of test case
	Avoid eye jumps
	Group similar test cases
	Coherent formatting
	Semantic structure (AAA, GWT, etc.)
Test names (6 vs 26)	
Use of patterns	Use of patterns
	Consistent naming
	Long names, spaces in names
	Include method under test in name
Assertions (5 vs 24)	
Number of assertions	One assert per test
Fluent assertions	Fluent assertions
Assertion messages	Assertion messages
	Appropriate assertions
	Custom assertions
Helper structures (0 vs 23)	
	Builder pattern
	Composition over inheritance (of test classes)
	Methods for each step (given when then)
	Page Objects
Dependencies (3 vs 19)	
One test for one behavior	One test for one behavior
Identifier names (5 vs 17)	
Consistent	Consistent
Concise	Concise
Meaningful	Meaningful
Use patterns	Use patterns
Fixtures (0 vs 17)	
	Use setup methods, avoid fixture
	No test data in fixtures
	Avoid long fixtures
DRY principle (0 vs 16)	
	Not too DRY, violate if needed
	Balance of DRY and DAMP

Table 5 continued

Scientific Literature	Grey Literature
Test data (4 vs 15	
No magic values	No magic values
Production like, typical, simple values	Production like, typical, simple values
	Hard coded expected values instead of computed
	Highlight important data
Comments (2 vs 14)	
Explanatory comments	Avoid, can become outdated
	Comments for structuring (e.g., AAA pattern)
Domain specific language (0 vs 8	
Explanatory comments	Avoid, can become outdated
	Comments for structuring (e.g., AAA pattern)
Parameterized test (0 vs 8	
	Data driven tests to reduce code duplication
Test summaries (4 vs 0)	
Use source code summarization techniques	
Textual features (1 vs 0)	
Natural language processing, dictionaries	

(Overlapping aspects such as recommendations and discussions are highlighted.)

Helper structures (0 vs 23) (Help): This factor has been *identified only in the grey literature*. In this context, the builder pattern and similar patterns are discussed, relating to practical recommendations for good design.

Dependencies (3 vs 19) (Dep): The recommendation that one test should only test one functionality or behavior is mentioned by Palomba et al. [A14].[5] The participants from the study of Setiani et al. [A17] to some extent agree that a unit test should only depend on one unit, which reflects the opinion of this factor from grey literature.

Identifier names (5 vs 17) (IdN): The survey from Lin et al. [A13] shows the importance of meaningful, concise and consistent identifiers. The renaming approach by Roy et al. [A16] also suggests variable names like *expected* and *result*. Their deep learning model was trained with software projects of a high level of quality. Therefore, it seems plausible that identifier names as those mentioned in the grey literature sources are commonly used in tests of high quality projects.

Fixtures (0 vs 17) (Fix): This factor has been *identified only in the grey literature*, which discusses arguments for and against the use of test fixtures from a practical perspective.

DRY principle (0 vs 16) (DRY): This factor has been *identified only in the grey literature*, n context of practical recommendations on how to apply this development principle to test code.

[5] The recommendation itself was proposed by Van Deursen et al. (2001).

Test data (4 vs 15) (TeD): Participants of the workshop from Bowes et al. [A4] also recommend to avoid magic values. Almasi et al. [A2] and Afshan [A1] highlight importance of meaningful or human-like test data.

Comments (2 vs 14) (Com): The usage of comments for highlighting the structure of the test is not investigated in scientific literature. Fisher and Johnson [A7] explain different readability ratings between generated tests and human tests also with the lack of explanatory comments. Setiani et al. [A17] survey participants who also mention comments being to some degree important to readability. These findings are to some extent contradicting the recommendation in grey literature, which is generally to avoid such explanatory comments.

Domain specific language (0 vs 8) (DSL): This factor has been *identified only in the grey literature*. It relates to test frameworks used in practice such as Gherkin.

Parameterized test (0 vs 8) (Par): This factor has been *identified only in the grey literature*. It relates to practical suggestions to reduce code duplication by using parameterized tests.

Test summaries (4 vs 0) (TS): This factor has been *identified only in the scientific literature*. It is related to the application of source code summarization techniques investigated in related research as support for understanding test code.

Textual features (1 vs 0) (TF): This factor has been *identified only in the scientific literature*. It is related to the application of natural language processing investigated in related research for test cases.

RQ2.2 Findings. What is the difference between influence factors in scientific literature and grey literature? 9 factors were identified in the scientific and 12 factors in the gray literature review, with an intersection of 7 factors identified in both. The factors identified most often in the scientific literature were also most frequently found in the grey literature: Test structures (5 and 28 times), test names (6 and 26 times), and assertions (5 and 24 times). Scientific and grey literature sources sometimes focus on different aspects of common factors.

6 Evaluation of Influence Factors

For the following experiment we take the results from the systematic mapping study (Section 4) and the grey literature review (Section 5) and investigate a selection of identified influence factors with focus on the perception of test case readability.

6.1 Experiment Setup and Procedure

The experiment follows an A/B testing approach. The participants rate readability of original and altered test cases. Experiments based on A/B testing are a good approach for comparing the effect of a treatment to a population. In our scientific literature review we also found some studies using this approach, e.g., Roy et al. (2020a) or Setiani et al. (2021a). Participants of a

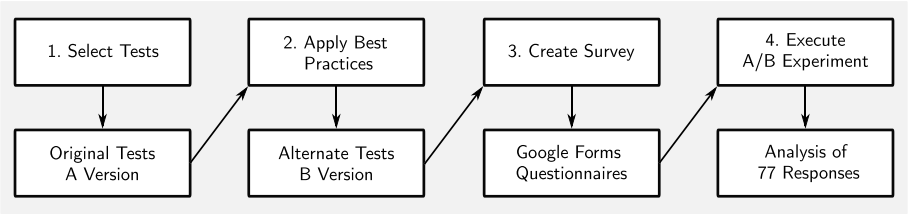


Fig. 6 Experiment process and amount of received responses

master course on software testing at TU Wien were invited to participate voluntarily in this online experiment with the possibility of bonus points for the course as a reward. Figure 6 shows an overview on the experiment process. We discuss the individual steps in the following sections.

6.1.1 Select Tests

We searched open source repositories for test case that are related to the influence factors we identified in our literature study, specifically test cases adhering or contradicting to these factors. We selected 30 test cases covering different influence factors from 8 sources, which also contain generated tests by Randoop and Evosuite. Table 6 shows influence factor, test name and origin project. Most tests including the automatically generated tests come from the open source project *Apache Commons Lang3*. Other sources include the *Spring Framework*, *IntelliJ*, and *Apache Flink*. The last three tests with origin project "Student Solution" are selected tests written by students for a course assignment.

The test cases we found in our search and which are used in the subsequent experiment cover 7 out of the 14 influence factors (see Table 4 in Section 5.2 for a complete list of influence factors), identified in the literature study, since we limited our selection to only those test cases retrieved from real-world open source projects that can be clearly related to individual influence factors. Therefore, we included test cases related to the influence factors *Structure (Str), Assertions (Asse), Dependencies (Depe), Test Data (TD), Comments (Co), Fixtures (Fix),* and *Parametrized Tests (Para)* and excluded test cases related to *Test Names (TeN), Identifyer Name (IdN), Test Summaries (TS), Textual Features (TF), Helper Structures (Help), DRY Principle (DRY),* and *Domain Specific Language (DSL).*

6.1.2 Apply Best Practices

For each test, we create an alternative version, following the findings from the literature study. For example, long test cases (variant A) were modified by splitting them up into two or more smaller test cases (variant B). This modification corresponds to the best practice suggested in [G28]. Similarly, test cases using standard assertions (variant A) were modified by using dedicated assertion frameworks such as AssertJ (variant B), as suggested in [G18].

Table 6 provides an overview of the covered influence factors from the literature study and a short description on the differences between A and B version of the different tests in column *"Modification A/B"*.

Furthermore, we used three additional test cases (not shown in the table) without modification as a control group. The purpose of these control tests is to verify that the participants show a consistent rating behavior for A and B tests, which allows us to assert the internal validity of the experiment.

Table 6 Listing of test cases with their assigned influence factor, originating project and differences made for both versions

Influence Factor	Test Name	Origin Project	Modification A/B
Structure	testPrimitiveTypeClass Serialization	Apache Commons Lang3	Loops vs. unrolled loops
Structure	testReducedFraction	Apache Commons Lang3	Loops vs. unrolled loops (exemplary values)
Structure	testContainsIgnoreCase _LocaleIndependence	Apache Commons Lang3	Loops vs. unrolled loops
Assertions	test10	Apache Commons Lang3	Try catch vs. AssertThrows
Assertions	test2	Apache Commons Lang3	Try catch vs. AssertThrows
Assertions	test303	Apache Commons Lang3	Try catch vs. AssertThrows
Structure	testInvert	Apache Commons Lang3	Variable reuse
Structure	testNegate	Apache Commons Lang3	Variable reuse
Structure	testAbs	Apache Commons Lang3	Variable reuse
Structure	test551	Apache Commons Lang3	Remove package names, if-structure, system out print and helper variables
Structure	test0074	Apache Commons Lang3	Remove package names, if-structure, system out print and helper variables
Structure	test1113	Apache Commons Lang3	Remove package names, if-structure, system out print and helper variables
Comment	testContainsRange	Apache Commons Lang3	Remove comments
Comment	testFactory_double	Apache Commons Lang3	Remove comments
Comment	testWrap_StringInt StringBooleanString	Apache Commons Lang3	Remove comments
Parameterized	testPrimitiveTypeClass Serialization	Apache Commons Lang3	Loops vs. Parameterized. Replace with @MethodSource
Parameterized	testReducedFraction	Apache Commons Lang3	Loops vs. Parameterized. Replace with @MethodSource (chained stream)
Parameterized	testContainsIgnoreCase _LocaleIndependence	Apache Commons Lang3	Loops vs. Parameterized. Replace with inlined CSV
Dependencies	testAllNullBooleans	Apache Flink	Split up tests
Dependencies	testSerializeAndParse	Protoclbuffers Protobuf	Split up tests (original has comments)
Dependencies	testSetContentObject	Apache Commons Email	Split up tests (original has comments)
Assertions	testFourElement2	JetBrains IntelliJ Community	Specific assertions (JUnit vs. Hamcrest/AssertJ)

Table 6 continued

Influence Factor	Test Name	Origin Project	Modification A/B
Assertions	showsAllStsGaDownloads	Dchartfield Sagan	Specific assertions (JUnit vs. Hamcrest/AssertJ)
Assertions	indexedReadAndIndexed WriteMethods	Spring Framework	Specific assertions (JUnit vs. Hamcrest/AssertJ)
Structure	testChoicesWithValid DefaultValue	Apache Flink	Remove unnecessary try catch
Structure	testApplyToMovesValue PassedOnShortName ToLongNameIfLong NameIsUndefined	Apache Flink	Remove unnecessary try catch
Structure	testApplyToWithMultiple Types	Apache Flink	Remove unnecessary try catch
Fixture, test data	Student Example 03	Student Solution	Remove fixture and member variables or constants
Fixture, test data	Student Example 02	Student Solution	Remove fixture and member variables or constants or constants
Fixture, test data	Student Example 01	Student Solution	Remove fixture and member variables or constants

A (original version) and **B** (altered version) denote the groups

6.1.3 Create Survey

We created surveys containing a subset of 12 test cases out of the entire set of the 30 tests listed in Table 6. Each survey contained an equal mix of A and B variants. In total we created 6 different surveys to provide full coverage of all 30 tests in each of the variants. The surveys were randomly distributed to the participants taking part in the experiment, who were unaware of the covered influence factors and whether the included tests were modified or unmodified.

The participants were asked to rate the readability on a 5 point Likert scale from 1 (unreadable) to 5 (easy to read) and to provide up to three free text reasons for their rating. Before and after this main task of the experiment, there is a pre- and post-questionnaire for collection information on the participants' background and feedback about the experiment run.

We developed the questionnaires using *google.forms*, which provides an easy way for creating surveys that can also be reused for future replications. The collected data can be exported in various formats for processing and analysis. Beside the survey forms, we provided the selected tests in a PDF and as plain text files as additional supporting materials for the study participants.

6.1.4 Execute A/B Experiment

The online survey was open for two weeks and the participants were free to start and stop their run at any time in this period. The duration for taking part in the experiment was about one hour per participant. In total, 77 participants completed the survey.

6.1.5 Analysis

We use the software R to calculate the significance of the results with statistical tests on level of $\alpha = 0.05\%$. According to an analysis with the Shapiro-Wilk test, the rating data does not follow a normal distribution. Therefore, and since our data is unpaired, we use the Wilcoxon Rank Sum test. When a significant difference between the distribution of the groups **A** and **B** is found, we report the effect size with Cliff's Delta (δ). Roy et al. (2020a) used the same approach for their Likert scale data. Cliff's Delta is interpreted according to Romano et al. (2006) with $|\delta| < 0.147$ *"negligible"*, $|\delta| < 0.33$ *"small"*, $|\delta| < 0.474$ *"medium"*, otherwise *"large"*.

6.2 Experiment Results

This section presents the results of the experiment on the readability of the selected set of test cases to investigate the related influence factors. Some factors influencing readability appear more than once in Table 6 and the modifications have different goals. Therefore we analyse the differences between groups A and B across these modifications. We discuss each modification after an overview on the participants in the following sections.

Participants experience. To gather some information about our participants we asked for their amount of experience in general and professional software development in years. They could choose between 0, 1-2, 2-5 and >5 years. Table 7 shows results of both questions. Almost 45% of our participants have more than five years of experience in software development and more than 50% have two to five years of experience. Concerning professional development around 30% have either one to two or two to five years of experience. In total around 75% have worked at least one year.

Table 7 Information on participants experience

Years	Absolute	Percentage
(a) General Software Development Experience [years]		
0	0	0%
1-2	2	2.6%
2-5	41	53.2%
>5	34	44.2%
Sum:	77	100.0%
(b) Professional Software Development Experience [years]		
0	20	26.0%
1-2	24	31.2%
2-5	25	32.5%
>5	8	10.3%
Sum:	77	100.0%

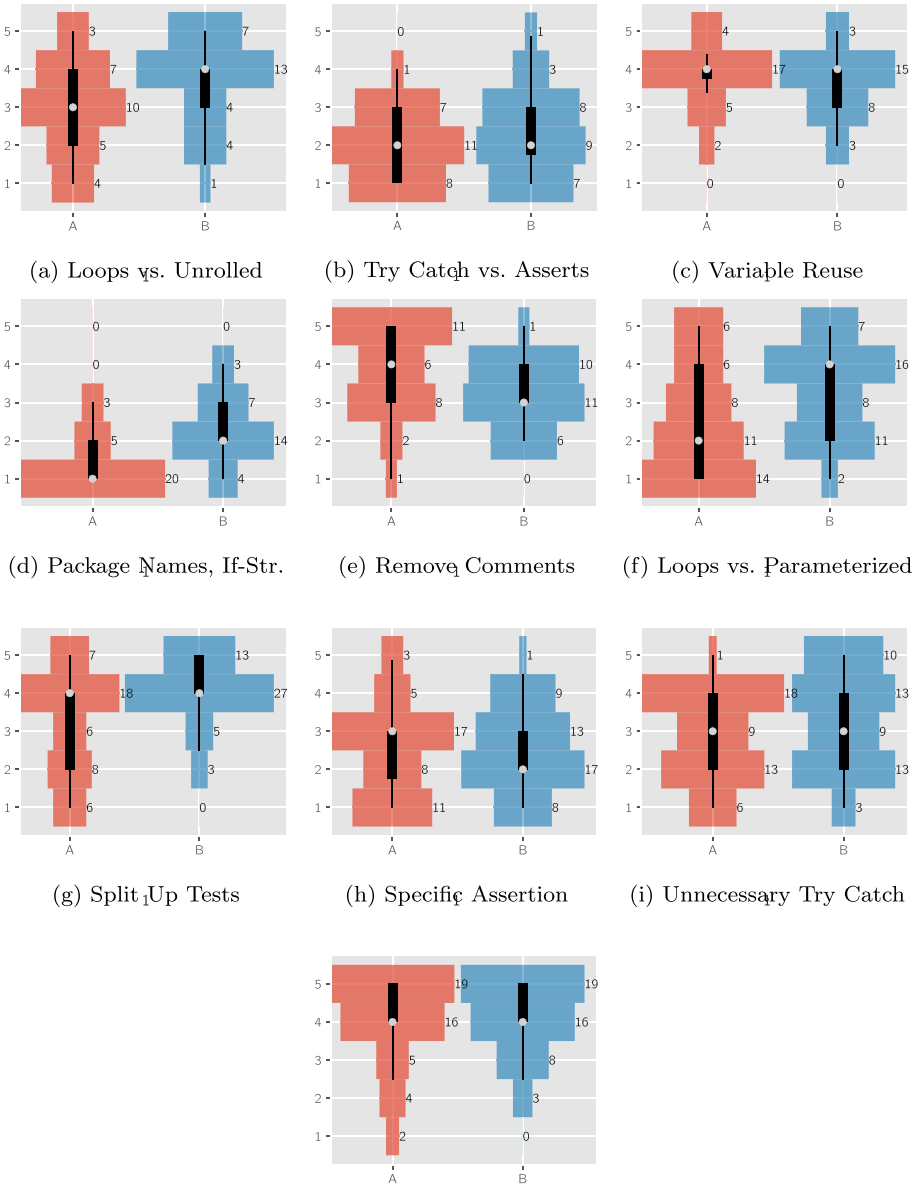
6.2.1 Do factors discussed in practice show an influence on readability when scientific methods are used (RQ3.1)?

Figure 7 shows the distribution of the aggregated readability ratings including boxplots for the investigated modification mapping to influence factors. Table 8 shows the results from the statistical analysis. The first column "Modification A/B (Influence Factor) maps to the according columns in Table 6. We discuss each modification in the following sections. As a reminder, we interpret Cliff's Delta (δ) according to Romano et al. (2006) with $|\delta| < 0.147$ *"negligible"*, $|\delta| < 0.33$ *"small"*, $|\delta| < 0.474$ *"medium"*, otherwise *"large"*,

Loops vs. Unrolled (Fig. 7a). In this modification the difference between A and B of the aggregated results is significant with $p = 0.02$. The effect size $\delta = -0.35$ is on the lower end of a *"medium"* effect size. The analysis of the individual tests reveals that the whole modification is significant, because of the last test with $p = 0.01$ and $\delta = -0.67$ (*"large"* effect). In this test the code contains two 2D arrays, nested loops to perform the test and string concatenation for the assertion message. The modified version primarily consists of assertions for all cases the loops generate, without assertion messages.

Try Catch vs. AssertThrows (Fig. 7b). Overall there is no significant difference in the readability ratings between the original and the modified versions. Only in the second test the difference between A and B is barely significant $p = 0.04$, although it has a *"large"* effect size with $\delta = -0.54$. One possible explanation for this result could be the relative short size of this test in comparison to the other ones in this modification. Due to the short length, there may be no possibility for other bad practices to mask the positive influence of this modification.

Variable Re-Use (Fig. 7c). Neither the figure nor the statistical analysis show a significant difference in the ratings.



(a) Loops vs. Unrolled (b) Try Catch vs. Asserts (c) Variable Reuse

(d) Package Names, If-Str. (e) Remove Comments (f) Loops vs. Parameterized

(g) Split Up Tests (h) Specific Assertion (i) Unnecessary Try Catch

(j) Remove Fixture

Fig. 7 Distribution and box plots of aggregated readability ratings per A/B modification. Ratings from a five-point Likert scale range from 1 (not readable) to 5 (very readable). The numbers on the right hand side of the histograms represent the amount of answers for this rating

Table 8 Statistical analysis of experiment results using a two-sided Wilcoxon Rank Sum test (p) and Cliff's D (δ) for effect size

Modification A/B (Influence Factor)	A N	med	sd	B N	med	sd	Compare p	δ
Loops vs. Unrolled Loops (Structure)	29	3	1.2	29	4	1.1	**0.02**	**-0.35**
testPrimitiveTypeClassSerialization	9	4	0.8	11	4	0.9	0.21	
testReducedFraction	9	3	0.9	9	3	1.3	0.89	
testContainsIgnoreCase_LocaleIndependence	11	2	1.2	9	4	0.9	**0.01**	**-0.67**
Try Catch vs. AssertThrows (Assertions)	27	2	0.9	28	2	1.1	0.31	
test10	9	2	0.9	10	2	1.2	0.90	
test2	9	2	0.8	9	3	1.1	**0.04**	**-0.54**
test303	9	2	1	9	2	0.9	0.89	
Variable Reuse (Structure)	28	4	0.8	29	4	0.8	0.32	
testInvert	9	4	0.7	11	3	0.9	0.54	
testNegate	9	4	0.9	9	4	0.5	0.15	
testAbs	10	4	0.7	9	4	0.9	0.78	
Package Names, If-Structure,.. (Structure)	28	1	0.7	28	2	0.9	**0.00**	**-0.59**
test551	8	1	0.7	11	3	0.8	**0.00**	**-0.82**
test0074	9	1	0.9	8	2	0.8	0.16	
test1113	11	1	0.5	9	2	0.4	**0.03**	**-0.51**
Remove Comments (Comments)	28	4	1.1	28	3	0.8	**0.02**	**0.36**
testContainsRange	9	4	0.9	10	4	0.7	0.43	
testFactory_double	9	5	1.1	9	3	0.7	0.08	
testWrap_StringIntStringBooleanString	10	4	1.4	9	2	0.9	0.15	
Loops vs. Parameterized (Parameterized)	45	2	1.4	44	4	1.2	**0.00**	**-0.34**
testPrimitiveTypeClassSerialization	14	4	1.1	15	4	0.9	0.85	
testReducedFraction	14	2	1.4	14	2	1.2	0.77	
testContainsIgnoreCase_LocaleIndependence	17	1	0.8	15	4	1.1	**0.00**	**-0.84**
Split Up Tests (Dependencies)	45	4	1.3	48	4	0.8	**0.00**	**-0.33**
testAllNullBooleans	14	4	1.4	17	4	0.7	0.08	
testSerializeAndParse	15	4	1.3	16	4	0.9	0.60	
testSetContentObject	16	3	1.2	15	4	0.8	**0.01**	**-0.50**
Specific Assertion (Assertions)	44	3	1.2	48	2	1.1	0.93	
testFourElement2	16	3	1	17	2	1.2	0.44	
showsAllStsGaDownloads	14	3	1.2	16	3	0.8	0.73	
indexedReadAndIndexedWriteMethods	14	2	1.3	15	3	1.1	0.47	
Unnecessary Try Catch (Structure)	47	3	1.1	48	3	1.3	0.12	
testChoicesWithValidDefaultValue	16	4	0.9	17	4	1.1	0.90	
testApplyToMovesValuePassedOnShortName...	15	2	1.1	16	2	1.1	0.90	
testApplyToWithMultipleTypes	16	2.5	1.1	15	4	1.2	**0.01**	**-0.53**
Remove Fixture (Fixture, Test Data)	46	4	1.1	46	4	0.9	0.87	
Student Example 03	16	4	1.3	16	4	1.1	0.54	
Student Example 02	15	4	0.9	15	5	0.7	0.17	
Student Example 01	15	5	0.9	15	4	0.8	0.10	

δ is only shown for $p < 0.05$

Structure (Fig. 7d). Overall there is a clear difference between the groups of this modification with $p = 0.0$ and a *"large"* effect, $\delta = -0.59$. Only for one of the three tests the difference between groups is not significant with $p = 0.16$.

Comments (Fig. 7e). Although none of the individual tests has a significant difference between A and B, the aggregated result is significantly different with $p = 0.02$ and has a lower *"medium"* effect size with $\delta = 0.36$. Since we removed comments in the original versions of the tests, the A version contains more information than B. A look at Fig. 7e and the median values in the Table 8 shows that the participants gave the A version better ratings. This is also reflected by the positive sign of the effect size. The comments do not highlight the structure of the test, they are of the nature "explanatory comments". This is a confirmation of the positive influence of comments on readability found by scientific literature.

Loops vs Parameterized (Fig. 7f). Like in *Loops vs. Unrolled* the difference of the complete modification between groups A and B is significant with $p = 0.00$ and $\delta = -0.34$, a lower *"medium"* effect size, because of the last test. The original version is the same as in *Loops vs. Unrolled* but the modified version extracts the test case data into an inlined CSV as input for the parameterized test case. The other forms of parameterized tests did not lead to significant changes in the readability ratings. In pursuit of the hypothesis from Section 7.2 we also compare the ratings of this A group with the A group from *Loops vs. Unrolled*. When looking at the median values the hypothesis seems to hold, because the values from this modification are lower in two of three tests. However, the Wilcoxon test does not detect a significant difference in the ratings with $p = 0.11$.

Split Up (Fig. 7g). There is a clearly significant difference between A and B with $p = 0.0$ but only a *"small"* effect size, although with $\delta = -0.33$ it is on the edge to a "medium" effect size. In detail there is one significant test $p = 0.01$ with $\delta = -0.50$, a large effect size. When looking at the median values and the figures, we see that both versions are quite readable but the modified tests have few to no ratings in the lower part of the readability scale.

Assertions (JUnit, Hamcrest, AssertJ) (Fig. 7h). There is no significant difference in readability when using standard JUnit assertions compared to assertions with Hamcrest or AssertJ assertions. This result confirms findings from Leotta et al. (2018a).

Unnecessary Try Catch (Fig. 7i). One test shows a significant difference with $p = 0.01$ and $\delta = -0.53$, a *"large"* effect size. With medians of 0.75 the first test is almost very readable in both versions. However, we accidentally introduced an error in the modified version (we declared a variable twice, which is not allowed in Java). In the comments the participants noticed this error, therefore this error might mask the positive effect of the intended modification. The second test with medians of 0.25 has a very long test name which the participants criticise. This again might mask the positive effect of the modification.

Fixture (Fig. 7j). We do not see a significant difference between the two versions neither in the figure nor in the table. The tests all have a quite good rating, which is could be caused by the participants knowledge about the system under test.

RQ3.1 Findings. Do factors discussed in practice show an influence on readability when scientific methods are used? Applying industry best practices is no silver bullet for improving the readability of test code. In the scientific experiments, the modifications showed a statistically significant positive influence on the readability of the tests for 50% of the factors, i.e., in five out of ten cases.

7 Summary, Threats to Validity, and Future Work

This section summarizes the findings, validity, presents implications for research and practitioners, and provides discusses limitations and threats to validity, and provides future research directions.

7.1 Summary

The main goal of this paper was to combine scientific and practical views on the readability of software test code. We have conducted a Systematic Mapping Study (SMS) to cover relevant publications from academia to capture the scientific view on the readability of software test code. We have complemented the results of the SMS by taking into consideration practical views based on grey literature. Based on identified influence factors on test code, we conducted a controlled experiment in academic setting to explore the perception of software test code readability with a set of 77 participants.

We have identified *unique readability factors in scientific literature* that include readability models, application of code summaries used on test code that have been proposed and evaluated (see Section 4.2.1).

Individual Influence Factors have been evaluated in scientific literature by using scientific methods, such as online questionnaires with Likert scales and statistical analysis (see Section 4.2.2).

Differences in scientific and grey literature. In contrast to scientific literature grey literature provides a wide spectrum of best practices and guidelines concerning the influence factors of readability of test code. There is large overlap between science and grey literature, however in the overlapping factors we observed different views in the interpretations. (see Section 5.2.2).

Unique readability factors in grey literature. Furthermore we found five additional factors exclusive to grey literature, e.g. helper structures, test fixtures. Concerning these factors there exist different views and even conflicting opinions, often related to the used/applied technology, testing framework, and test level/approach (see Section 5.2.1).

Empirical study of influence factors widely discussed in practice. For half of the investigated modifications (Loops vs. Unrolled Loops; Package Names, If-Structures; Remove Comments; Loops vs. Parameterized and; Split Up Tests), which map to readability factors, we could show a statistical significant influence in test code readability. (see Section 6.2.1). Other factors are less clear, which can be attributed to the nature of best practices, which are sometimes only applicable in specific contexts and not in general (e.g., modification Try Catch vs. AssertThrows; (see Section 6.2.1).

7.2 Limitations and Threats to Validity

In this section, we summarize important limitations and threats to validity in context of the literature review (i.e., scientific and grey literature) and the empirical study.

Internal Validity

- In context of the Systematic Mapping study, the keyword, search string, analysis items, and the data extraction and analysis has been executed by one of the authors and intensively reviewed and discussed within the author team and external experts.
- The controlled experiment setup has been initially executed in a pilot run to ensure consistency of the experiment material. We have used a cross-over design of test case samples to avoid any bias of the experiment participants.
- Three unmodified test cases were used as control groups in A/B testing. The Wilcoxon Rank Sum test does not suggest a significant difference between ratings provided by participants, when comparing groups with the same questionnaire. However, there is a significant effect when comparing control groups of different questionnaires. These results confirm a consistent rating behavior within groups and the significant differences between groups is as expected due to the independent ratings of participants from different groups.

External Validity

- We have conducted a literature reviews based on the guidelines of Petersen et al. (2015) complemented by a systematic analysis of grey literature (Garousi et al. 2019). Therefore, the analysis results identified most prominent research directions in scientific literature complemented by practical discussions in non-academic sources (such as blogs). This approach enabled us to identify similar and/or different key topics in academia and industry.
- Experiment participants were recruited on a voluntary basis from three classes of a master course on *software testing* at TU Wien. We captured background knowledge of the participants to identify participant experience. Most of the participants work in industry and can be considered as "junior professionals". Therefore, the results are applicable for industry applications.
- We used real-world test cases from open source projects as well as results from software testing exercises to ensure close to industry test cases.

Construct Validity

- We build on best-practices for the literature review for academic publications (Petersen et al. 2015) and grey literature (Garousi et al. 2019) and followed experimentation guidelines, proposed by Wohlin et al. (2012) for conducting the empirical study.
- For the controlled experiment, we captured individual test case assessments for A-B tests (i.e., original tests taken from existing projects and slightly modified test cases) based on a 5-point Likert scale.
- To avoid a bias introduced by the order of questions for the experiment, we reversed the question ordering for half of the experiment groups.
- To avoid random readability ratings, we asked participants to give reasons for their ratings as free text. Furthermore, the participants were told that their reward (bonus points) is coupled with active participation in the challenge.

– We tried to select test cases for A/B testing in our experiment, which could be clearly related to individual influence factors. Since the test cases we used were retrieved from real world projects instead of constructed examples, which could limit the relevance of our results, we only covered 7 out of the 14 influence factors identified in the literature search. Nevertheless, a certain amount of fuzziness with respect to influence factors may still be present, e.g., as discussed in the results for the modification Try Catch vs. AssertThrows (see Section 6.2.1).

Conclusion Validity

– We used the Shapiro-Wilk test for testing for normality, which would allow us to use a parametric statistical test. This approach is also used by Roy et al. (2020a) whose methodology is similar to ours.
– We used the non parametric Wilcoxon Rank Sum test, because our groups are unpaired and the Shapiro-Wilk test does not suggest a normal distribution of our result data.
– We report the effect size with Cliff's Delta, because it allows an interpretation of the magnitude of difference between two groups. It is also used by other studies in this field like Grano et al. (2018a).

7.3 Implications for Research and Practitioners

This section summarizes the main implications of the SMS (Section 4), the grey literature study (Section 5), and the experiment (Section 6).

Implications for Research

– For the *Software Testing* community, we identified influencing factors, observed only in grey literature, that could initiate additional research initiatives with focus on topics that are of interest for practitioners with limited attention of researchers.
– Researchers in *Software Engineering* and/or *Software Testing* can take up the results from literature review with focus on replicating and extending the presented research work.
– The *Empirical Research* community can build on the the SMS protocol, the grey literature protocol, and the study design to replicate and extend the study protocol in different context.
– We selected a representative set of test cases that could be used by researchers to (i) design and develop a method and or tool to semi-automatically assess the readability of test code and (ii) to apply the test code set for evaluation purposes in different contexts.
– In the *Software Testing* communities, factors, such as *Setup methods/Fixtures*, *Helper Methods*, *DRYness* are widely discussed in the domain of practitioners. Considering these in test code generation could be useful for generating more readable tests. In a recent study Panichella et al. (2022) also suggest to include capabilities for complex object instantiation into test suite generators.
– Finally, the findings of the study can be used as input for researchers from *Software Engineering* communities to improve software maintenance tasks that benefit from readability assessments.

Implications for Practitioners

- For *Software and System Engineering* organizations, results of this work can support software testers and developers to improve test code readability based on guidelines and identified influencing factors.
- *Project and Quality Managers* can use the results to setup organization specific development guidelines to support software development, software testing, and software maintenance and evolution by a team of software experts. Applied best practices might help to improve the quality of test cases and reduce effort and cost for maintenance activities.
- Factors with similar views from practitioners and academia include *Test Names*, *Identifier Names*, and *Test Data*. For test and identifier names both domains agree on the use of naming patterns in order to achieve consistency across the test suite. For test data also both domains agree on the use of realistic and simple values and avoiding magic values.
- However, the experiment results show that application of best-practices is no guarantee for improved readability.

7.4 Future Work

The main goal of this article was to **Investigate the readability of Software Test Code** by combining scientific and practical views. We applied a systematic mapping study for analyzing scientific literature complemented by grey literature. Furthermore, we executed a controlled experiment in a Software Testing Master Course on academic level to investigate practical implications of a selected but typical set of test cases.

In the future, we plan to replicate the experiment to increase the external validity of the study in academia, complemented by industry participants. Furthermore, we plan to develop and evaluate a maturity model for the readability of test cases that could help quality managers, software test engineers, and software developer in better assessing the quality of test cases (from readability perspective) to improve software maintenance and evolution.

A) Sources of Grey Literature for Systematic Mapping

[G1] M. Aniche. Effective Software Testing Version 4 - Academia.edu. nan, 2022. https://www.academia.edu/70086528/Effective%5C_Software%5C_Testing%5C_Version%5C_4
[G2] G. Bahmutov. Readable Cypress.io tests - Gleb Bahmutov. Blog. 2019. https://glebbahmutov.com/blog/readable-tests/
[G3] A. Bansal. Best Practices For Unit Testing In Java - Baeldung. Magazine. 2021. https://www.baeldung.com/java-unit-testingbest-practices
[G4] V. V. C. blog. The Unit Test Strategy In Vald. Blog. 2021. https://vdaas-vald.medium.com/the-unit-test-strategy-in-vald-912ed6f14fbd
[G5] P. Bloomfield. How to write a good unit test - Peter Bloomfield. Blog. 2020. https://peter.bloomfield.online/how-to-write-agood-unit-test/
[G6] R. Borowiec. Test code readability improved: JUnit with Mockito and FEST ... Blog. 2013. https://blog.codeleak.pl/2013/07/test-code-readability-improved-junit.html
[G7] M. Brizeno. Write Better Tests in 5 Steps - Thoughtworks. Blog. 2014. https://www.thoughtworks.com/insights/blog/write-better-tests-5-steps

[G8] V. Bulavin. Vadim Bulavin auf Twitter: ”9. A good unit test must have three ... Blog. 2020. https://twitter.com/v8tr/status/1217483476406079491?lang=de
[G9] D. Carter. Make your automated tests easy to read - DC Coding - Dan... Blog. 2011. https://codingblog.carterdan.net/2020/02/11/make-your-automated-tests-easy-to-read/
[G10] R. Casadei. Effective unit testing - SlideShare. Presentation. 2013. https://de.slideshare.net/RobertoCasadei/effective-unittesting
[G11] H. Coles et al. Write Damp Test Code - Java for small teams - ncrcoe. Wiki. 2015. https://ncrcoe.gitbooks.io/java-for-smallteams/content/v/restructure/tests/1900%5C_write%5C_damp%5C_test%5C_code.html
[G12] testing company. Unit Testing Tutorial: 5 Best Practices - VTEST Blog. Blog. 2020. https://www.vtestcorp.com/blog/unittesting-best-practices/
[G13] F. nickname Company blog. More readable tests with Kotlin - Tengio. Blog. 2016. https://www.tengio.com/blog/more-readabletests-with-kotlin/
[G14] R. Cook. In Programming, is it worth to sacrifice some DRY-ness for... Other. 2017. https://www.quora.com/In-Programming-isit-worth-to-sacrifice-some-DRY-ness-for-code-readabilityor-the-other-way-around
[G15] F. C. Correa. How to write legible QA tests - Codacy - Blog. Blog. 2019. https://blog.codacy.com/how-to-write-legible-qatests/
[G16] A. Dhoke. Do you really want moist test - AmeyDhoke’s blog. Blog. 2009. http://maverick-amey.blogspot.com/2009/05/doyou-really-want-moist-test.html
[G17] E. Dietrich. Test Readability: Best of All Worlds - DaedTech. Blog. 2013. https://daedtech.com/test-readability-best-ofall-worlds/
[G18] B. Dijkstra. Three practices for creating readable test code. Blog. 2016. https://www.ontestautomation.com/three-practices-forcreating-readable-test-code/
[G19] M. Duiker. Improving unit test readability: helper methods & named... Blog. 2016. https://blog.marcduiker.nl/2016/06/01/improving-unit-test-readability-named-args.html
[G20] T. F. Dustin Boswell. The Art of Readable Code. https://www.oreilly.com/library/view/the-art-of/9781449318482/ch14.html. O’Reilly, 2012
[G21] U. Enzler. Clean Code Cheat Sheet - planetgeek.ch. Cheatsheet. 2014. https://www.planetgeek.ch/wp-content/uploads/2014/11/Clean-Code-V2.4.pdf
[G22] J. Fernandes. Rethinking Testing Through Declarative Programming. Blog. 2020. https://betterprogramming.pub/rethinkingtesting-through-declarative-programming-335897703bdd
[G23] J. Flournoy. Category: coverage - Pervasive Code. Blog. 2008. http://www.pervasivecode.com/blog/category/coverage/
[G24] M. Foord. 30 best practices for software development and testing. Magazine. 2017. https://opensource.com/article/17/5/30-best-practices-software-development-and-testing
[G25] T. Goeschel. Writing Better Tests With JUnit - codecentric AG Blog. Blog. 2016. https://blog.codecentric.de/en/2016/01/writing-better-tests-junit/
[G26] J. Gorman. Readable Parameterized Tests - Codemanship’s Blog. Blog. 2020. https://codemanship.wordpress.com/2020/09/26/readable-parameterized-tests/
[G27] H. Grigg. A simple, readable, meaningful test style with Jest. Blog. 2020. https://notestoself.dev/posts/simple-readablemeaningful-jest-test-style/
[G28] P. Hauer. Modern Best Practices for Testing in Java - Philipp Hauer’s Blog. Blog. 2021. https://phauer.com/2019/modern-bestpractices-testing-java/
[G29] E. Haus. 102. Whether or Not to Repeat Yourself: DRY, DAMP, or WET. Podcast / transcript. 2020. https://www.heroku.com/podcasts/codeish/102-whether-to-repeat-yourself-dry-dampor-wet

[G30] B. Hnat. DRYer Tests - The Dumpster Fire Project. Blog. 2020. https://thedumpsterfireproject.com/dryer-tests
[G31] A. Huttunen. How to Make Your Tests Readable - Arho Huttunen. Blog. 2021. https://www.arhohuttunen.com/test-readability/
[G32] J. Jarrett. Fluent Specification Extensions - Developing on Staxmanade. Blog. 2009. https://staxmanade.com/2009/02/fluent-specification-extensions/
[G33] K. Johnson. Is duplicated code more tolerable in unit tests? - Stack Overflow. Stackoverflow. 2008. https://stackoverflow.com/questions/129693/is-duplicated-code-more-tolerable-inunit-tests
[G34] P. Kainulainen. Writing Clean Tests - Petri Kainulainen. Blog. 2014. https://www.petrikainulainen.net/writing-clean-tests/
[G35] T. Kareinen. Readable tests - Tuomas Kareinen's blog. Blog. 2012. https://tkareine.org/articles/readable-tests.html
[G36] V. Khorikov. DRY vs DAMP in Unit Tests - Enterprise Craftsmanship. Blog. 2008. https://enterprisecraftsmanship.com/posts/dry-damp-unit-tests/
[G37] L. Koskela. Effective Unit Testing. https://livebook.manning.com/effective-unit-testing/chapter-4. Manning, 2013.
[G38] A. Lal. Kotlin DSL - let's express code in "mini-language" - Part 5 of 5. Blog. 2019. https://www.aditlal.dev/kotlin-dsl-part-5/
[G39] D. Lehner. 3 easy fixes for perfect unit test code - devmate. Blog. 2021. https://www.devmate.software/3-easy-fixes-forperfect-unit-test-code/
[G40] D. Lindner. unit test - Schneide Blog. Blog. 2013. https://schneide.blog/tag/unit-test/
[G41] P. Lipinski. or how to write tests so that they serve you well. Presentation. 2013. https://2013.jokerconf.com/presentations/03%5C_02%5C_lipinski%5C_pawel%5C_jokerconf-presentation.pdf
[G42] N. MANJUNATH and O. D. MEULDER. No Code? No Problem - Writing Tests in Plain English - NYT ... Blog. 2019. https://open.nytimes.com/no-code-no-problem-writing-tests-inplain-english-537827eaaa6e
[G43] R. C. Martin. Clean Code: Chapter 9. https://reee3.home.blog/2021/02/17/clean-code-9/. nan, 2021.
[G44] B. Myers. Readable test code matters. - Brooklin Myers. Blog. 2021. https://brooklinmyers.medium.com/readable-test-codematters-e46cc5c411bb
[G45] M. Needham. TDD: Test DRYness - Mark Needham. Blog. 2009. https://www.markhneedham.com/blog/2009/01/30/tdd-testdryness/
[G46] T. Papendieck. Why Sometimes Unit Tests do More Harm than Good? Blog. 2017. https://www.beyondjava.net/why-sometimesunit-tests-do-more-harm-than-good
[G47] C. Pip. Clean Code in Tests: What, Why and How? - TestProject. Blog. 2020. https://blog.testproject.io/2020/04/22/cleancode-in-tests-what-why-and-how/
[G48] P. Reagan. Keep Your Friends Close, But Your Test Data Closer -Viget. Blog. 2009. https://www.viget.com/articles/keepyour-friends-close-but-your-test-data-closer/
[G49] J. Reid. 3 Reasons Why It's Important to Refactor Tests - Quality Coding. Blog. 2016. https://qualitycoding.org/why-refactortests/
[G50] J. Roberts. Diagnosing Failing Tests More Easily and Improving Test... Blog. 2019. http://dontcodetired.com/blog/post/Diagnosing-Failing-Tests-More-Easily-and-Improving-Test-Code-Readability
[G51] J. Roberts. Improve Test Asserts with Shouldly - Visual Studio Magazine. Magazine. 2015. https://visualstudiomagazine.com/articles/2015/08/01/improve-test-asserts-with-shouldly.aspx.

[G52] M. Rodrigues. What Makes Good Unit Test? Readability - Matheus Rodrigues. Blog. 2018. https://matheus.ro/2018/01/15/makes-good-unit-test-readability/
[G53] J. V. Ryswyck. Avoid Inheritance For Test Classes - Principal IT. Blog. 2021. https://principal-it.eu/2021/01/avoid-inheritance-for-test-classes/
[G54] A. Sarna. Do you think your code is Perfect? Well, Think again. Blog. 2018. https://blog.knoldus.com/do-you-think-your-codeis-perfect-well-think-again/
[G55] S. Scalabrino. "Automatically Assessing and Improving Code Readability and Understandability". https://iris.unimol.it/retrieve/handle/11695/90885/92359/Tesi_S_Scalabrino.pdf. PhD thesis. Universit'a degli Studi del Molise, 2019.
[G56] C. Schults. Unit Testing Best Practices: 9 to Ensure You Do It Right. Blog. 2021. https://www.testim.io/blog/unit-testingbest-practices/
[G57] J. Shih. A Field Guide to Unit Testing: Readability. Blog. 2020. https://codecharms.me/posts/unit-testing-readability
[G58] J. F. Smart. What makes a great test automation framework? - LinkedIn. Blog. 2020. https://www.linkedin.com/pulse/what-makesgreat-test-automation-framework-john-ferguson-smart
[G59] D. Snyder and E. Kuefler. Testing on the Toilet: Tests Too DRY? Make Them DAMP! Blog. 2019. https://testing.googleblog.com/2019/12/testing-on-toilet-tests-too-dry-make.html
[G60] S. Vance. Quality Code: Software Testing Principles, Practices, And ... https://vdoc.pub/documents/quality-code-software-testingprinciples-practices-and-patterns-781g7idb34f0. nan, 2013.
[G61] T. Yonekubo. Readable Test Code - Medium. Blog. 2021. https://medium.com/@t-yonekubo/readable-test-code-cad8a7babc7b
[G62] G. Zilberfeld. Test Attribute #2: Readability - Java Code Geeks. Blog. 2014. https://www.javacodegeeks.com/2014/07/test-attribute-2-readability.html

B) Sources of Academic Literature for Systematic Mapping

[A1] S. Afshan, P. McMinn, and M. Stevenson. "Evolving readable string test inputs using a natural language model to reduce human oracle cost". In: Proceedings - IEEE 6th Int. Conf. on Software Testing, Verification and Validation, ICST 2013. 2013, pp. 352-361.
[A2] M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. "An industrial evaluation of unit test generation: Finding real faults in a financial application". In: Proceedings - 2017 IEEE/ACM 39th Int. Conf. on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 263-272.
[A3] A. Alsharif, G. M. Kapfhammer, and P. McMinn. "What Factors Make SQL Test Cases Understandable for Testers? A Human Study of Automated Test Data Generation Techniques". In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2019, pp. 437-448. https://doi.org/10.1109/ICSME.2019.00076
[A4] D. Bowes, T. Hall, J. Petrić, T. Shippey, and B. Turhan. "How Good Are My Tests?" In: Int. Workshop on Emerging Trends in Software Metrics, WETSoM. IEEE Computer Society, 2017, pp. 9-14.
[A5] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. "Modeling readability to improve unit tests". In: 2015 10th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on the Foundations of Software Engineering,

ESEC/FSE 2015 - Proceedings. Association for Computing Machinery, Inc, 2015, pp. 107-118.
[A6] E. Daka, J. Rojas, and G. Fraser. "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" In: ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT Int. Symposium on Software Testing and Analysis. Association for Computing Machinery, Inc, 2017, pp. 57-67.
[A7] G. Fisher and C. Johnson. "Specification-Based testing in software engineering courses". In: SIGCSE 2018 - Proc. of the 49th ACM Techn. Symposium on Computer Science Education. Vol. 2018-January. Association for Computing Machinery, Inc, 2018, pp. 800-805.
[A8] G. Fraser and A. Zeller. "Exploiting common object usage in test case generation". In: Proceedings - 4th IEEE Int. Conf. on Software Testing, Verification, and Validation, ICST 2011. 2011, pp. 80-89.
[A9] G. Grano, S. Scalabrino, H. Gall, and R. Oliveto. "An empirical investigation on the readability of manual and generated test cases". In: Proceedings of the Int. Conf. on Software Engineering. IEEE Computer Society, 2018, pp. 348-351.
[A10] M. Leotta, M. Cerioli, D. Olianas, and F. Ricca. "Fluent vs Basic Assertions in Java: An Empirical Study". In: 2018 11th International Conference on the Quality of Information and communications Technology (QUATIC). 2018, pp. 184-192. https://doi.org/10.1109/QUATIC.2018.00036
[A11] B. Li, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and N. Kraft. "Automatically Documenting Unit Test Cases". In: Proceedings - 2016 IEEE Int. Conf. on Software Testing, Verification and Validation, ICST 2016. Institute of Electrical and Electronics Engineers Inc., 2016, pp. 341-352.
[A12] B. Li, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. "Aiding Comprehension of Unit Test Cases and Test Suites with Stereotype- Based Tagging". In: Proceedings of the 26th Conference on Program Comprehension. 2018, pp. 52-63. doi: 10.1145/3196321.3196339.
[A13] B. Lin, C. Nagy, G. Bavota, A. Marcus, and M. Lanza. "On the quality of identifiers in test code". In: Proceedings - 19th IEEE Int. Working Conf. on Source Code Analysis and Manipulation, SCAM 2019. Institute of Electrical and Electronics Engineers Inc., 2019, pp. 204-215.
[A14] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. "Automatic test case generation: What if test code quality matters?" In: ISSTA 2016 - Proceedings of the 25th Int. Symposium on Software Testing and Analysis. Association for Computing Machinery, Inc, 2016, pp. 130-141.
[A15] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. Gall. "The impact of test case summaries on bug fixing performance: An empirical investigation". In: Proceedings - Int. Conf. on Software Engineering. Vol. 14-22-May-2016. IEEE Computer Society, 2016, pp. 547-558.
[A16] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli. "DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests". In: Proceedings - 2020 35th IEEE/ACM Int. Conf on Automated Software Engineering, ASE 2020. Institute of Electrical and Electronics Engineers Inc., 2020, pp. 287-298.
[A17] N. Setiani, R. Ferdiana, and R. Hartanto. "Developer's Perspectives on Unit Test Cases Understandability". In: Proceedings of the IEEE Int. Conf. on Software Engineering and Service Sciences, ICSESS. Vol. 2021-August. IEEE Computer Society, 2021, pp. 251-255.
[A18] N. Setiani, R. Ferdiana, and R. Hartanto. "Test case understandability model". In: IEEE Access 8 (2020), pp. 169036-169046.

[A19] B. Zhang, E. Hill, and J. Clause. "Towards automatically generating descriptive names for unit tests". In: ASE 2016 - Proceedings of the 31st IEEE/ACM Int. Conf. on Automated Software Engineering. Association for Computing Machinery, Inc, 2016, pp. 625-636.

Acknowledgements This work has been partially supported via SBA Research (SBA-K1), a COMET Center within the COMET - Competence Centers for Excellent Technologies Programme, funded by BMK, BMAW, and the federal state of Vienna and managed by the Austrian Research Promotion Agency (FFG). This work has been partially supported via the COMET Center "Austrian Competence Center for Digital Production" (CDP) [881843] and the K2 centre InTribology [872176].This work has been partially supported by the Austrian Research Promotion Agency (FFG) in the frame of the project ConTest [888127] and via the COMET competence center INTEGRATE [892418] of SCCH, funded by BMK, BMAW, and the federal state of Upper Austria. Finally, the financial support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital & Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

Funding Open access funding provided by TU Wien (TUW).

Data Availibility Statement The datasets generated during and/or analysed during the current study are available via the TU Wien research data respository https://doi.org/10.48436/w4q8v-28695

Declaration

Conflicts of interest None

References

Afshan S, McMinn P, Stevenson M (2013) Evolving readable string test inputs using a natural language model to reduce human oracle cost. In: Proceedings - IEEE 6th Int. Conf. on Software Testing, Verification and Validation, ICST 2013, pp 352–361

Al Madi N (2022) How readable is model-generated code? examining readability and visual inspection of github copilot. In: 37th IEEE/ACM International conference on automated software engineering, pp 1–5

Almasi M, Hemmati H, Fraser G, Arcuri A, Benefelds J (2017) An industrial evaluation of unit test generation: Finding real faults in a financial application. In: Proceedings - 2017 IEEE/ACM 39th Int. Conf. on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Institute of Electrical and Electronics Engineers Inc., pp 263–272

Alsharif A, Kapfhammer GM, McMinn P (2019) What factors make sql test cases understandable for testers? a human study of automated test data generation techniques. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 437–448, https://doi.org/10.1109/ICSME.2019.00076

Aniche M (2022) Effective Software Testing Version 4 - Academia.edu. nan, https://www.academia.edu/70086528/Effective_Software_Testing_Version_4

Bahmutov G (2019) Readable cypress.io tests - gleb bahmutov. Blog, https://glebbahmutov.com/blog/readable-tests/

Bai GR, Presler-Marshall K, Fisk SR, Stolee KT (2022) Is assertion roulette still a test smell? an experiment from the perspective of testing education. In: 2022 IEEE Symposium on visual languages and human-centric computing (VL/HCC), pp 1–7, https://doi.org/10.1109/VL/HCC53370.2022.9833107

Bansal A (2021) Best practices for unit testing in java - baeldung. Magazine, https://www.baeldung.com/java-unit-testing-best-practices
Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2015) Are test smells really harmful? an empirical study. Empirical Software Engineering 20(4):1052–1094
blog VVC (2021) The unit test strategy in vald. Blog, https://vdaas-vald.medium.com/the-unit-test-strategy-in-vald-912ed6f14fbd
Bloomfield P (2020) How to write a good unit test - peter bloomfield. Blog, https://peter.bloomfield.online/how-to-write-a-good-unit-test/
Borowiec R (2013) Test code readability improved: Junit with mockito and fest ... Blog, https://blog.codeleak.pl/2013/07/test-code-readability-improved-junit.html
Bowes D, Hall T, Petrić J, Shippey T, Turhan B (2017) How good are my tests? IEEE Computer Society, Workshop on Emerging Trends in Software Metrics, Int. WETSoM, pp 9–14
Brizeno M (2014) Write better tests in 5 steps - thoughtworks. Blog, https://www.thoughtworks.com/insights/blog/write-better-tests-5-steps
Bulavin V (2020) Vadim bulavin auf twitter: "9. a good unit test must have three ... Blog, https://twitter.com/v8tr/status/1217483476406079491?lang=de
Buse RP, Weimer WR (2008) A metric for software readability. In: Proceedings of the 2008 int. symposium on Software testing and analysis, pp 121–130
Carter D (2011) Make your automated tests easy to read - dc coding - dan ... Blog, https://codingblog.carterdan.net/2020/02/11/make-your-automated-tests-easy-to-read/
Casadei R (2013) Effective unit testing - slideshare. Presentation, https://de.slideshare.net/RobertoCasadei/effective-unit-testing
Ceccato M, Marchetto A, Mariani L, Nguyen CD, Tonella P (2015) Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency https://doi.org/10.1145/2768829
Coles H, et al. (2015) Write damp test code - java for small teams - ncrcoe. Wiki, https://ncrcoe.gitbooks.io/java-for-small-teams/content/v/restructure/tests/1900_write_damp_test_code.html
Cook R (2017) In programming, is it worth to sacrifice some dry-ness for ... Other, https://www.quora.com/In-Programming-is-it-worth-to-sacrifice-some-DRY-ness-for-code-readability-or-the-other-way-around
Correa FC (2019) How to write legible qa tests - codacy - blog. Blog, https://blog.codacy.com/how-to-write-legible-qa-tests/
Daka E, Campos J, Fraser G, Dorn J, Weimer W (2015) Modeling readability to improve unit tests. In: 2015 10th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings, Association for Computing Machinery, Inc, pp 107–118
Daka E, Rojas J, Fraser G (2017) Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In: ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT Int. Symposium on Software Testing and Analysis, Association for Computing Machinery, Inc, pp 57–67
Deiß T (2008) Refactoring and converting a ttcn-2 test suite. Int Journal on Software Tools for Technology Transfer 10(4):347–352
Deissenboeck F, Wagner S, Pizka M, Teuchert S, Girard JF (2007) An activity-based quality model for maintainability. In: 2007 IEEE International conference on software maintenance, IEEE, pp 184–193
Delplanque J, Ducasse S, Polito G, Black AP, Etien A (2019) Rotten green tests. In: 2019 IEEE/ACM 41st International conference on software engineering (ICSE), IEEE, pp 500–511
Dhoke A (2009) Do you really want moist test - ameydhoke's blog. Blog, http://maverick-amey.blogspot.com/2009/05/do-you-really-want-moist-test.html
Dietrich E (2013) Test readability: Best of all worlds - daedtech. Blog, https://daedtech.com/test-readability-best-of-all-worlds/
Dijkstra B (2016) Three practices for creating readable test code. Blog, https://www.ontestautomation.com/three-practices-for-creating-readable-test-code/
Duiker M (2016) Improving unit test readability: helper methods & named ... Blog, https://blog.marcduiker.nl/2016/06/01/improving-unit-test-readability-named-args.html
Dustin Boswell TF (2012) The Art of Readable Code. O'Reilly, https://www.oreilly.com/library/view/the-art-of/9781449318482/ch14.html
Enzler U (2014) Clean code cheat sheet - planetgeek.ch. Cheatsheet, https://www.planetgeek.ch/wp-content/uploads/2014/11/Clean-Code-V2.4.pdf
Femmer H, Mund J, Fernández DM (2015) It's the activities, stupid! a new perspective on re quality. In: 2015 IEEE/ACM 2nd International workshop on requirements engineering and testing, IEEE, pp 13–19
Fernandes J (2020) Rethinking testing through declarative programming. Blog, https://betterprogramming.pub/rethinking-testing-through-declarative-programming-335897703bdd

Fisher G, Johnson C (2018) Specification-based testing in software engineering courses. In: SIGCSE 2018 - Proc. of the 49th ACM Techn. Symposium on Computer Science Education, Association for Computing Machinery, Inc, vol 2018-January, pp 800–805
Flournoy J (2008) Category: coverage - pervasive code. Blog, http://www.pervasivecode.com/blog/category/coverage/
Foord M (2017) 30 best practices for software development and testing. Magazine, https://opensource.com/article/17/5/30-best-practices-software-development-and-testing
Fraser G, Staats M, McMinn P, Arcuri A, Padberg F (2013) Does automated white-box test generation really help software testers? In: Proceedings of the 2013 International symposium on software testing and analysis, Association for Computing Machinery, ISSTA 2013, p 291-301, https://doi.org/10.1145/2483760.2483774
Fraser G, Zeller A (2011) Exploiting common object usage in test case generation. In: Proceedings - 4th IEEE Int. Conf. on Software Testing, Verification, and Validation, ICST 2011, pp 80–89
Garousi V, Felderer M (2016) Developing, verifying, and maintaining high-quality automated test scripts. IEEE Software 33(3):68–75
Garousi V, Küçük B (2018) Smells in software test code: a survey of knowledge in industry and academia. Journal of Systems and Software 138:52–81
Garousi V, Felderer M, Mäntylä MV (2019) Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Information and Software Technology 106:101–121
Goeschel T (2016) Writing better tests with junit - codecentric ag blog. Blog, https://blog.codecentric.de/en/2016/01/writing-better-tests-junit/
Gorman J (2020) Readable parameterized tests - codemanship's blog. Blog, https://codemanship.wordpress.com/2020/09/26/readable-parameterized-tests/
Grano G, De Iaco C, Palomba F, Gall HC (2020) Pizza versus pinsa: On the perception and measurability of unit test code quality. In: 2020 IEEE Int. conf. on software maintenance and evolution (ICSME), IEEE, pp 336–347
Grano G, Scalabrino S, Gall H, Oliveto R (2018a) An empirical investigation on the readability of manual and generated test cases. In: Proceedings of the int. conf. on software engineering, IEEE computer society, pp 348–351
Grano G, Scalabrino S, Gall H, Oliveto R (2018b) An empirical investigation on the readability of manual and generated test cases. In: Proceedings of the Int. Conf. on Software Engineering, IEEE Computer Society, pp 348–351
Grigg H (2020) A simple, readable, meaningful test style with jest. Blog, https://notestoself.dev/posts/simple-readable-meaningful-jest-test-style/
Hauer P (2021) Modern best practices for testing in java - philipp hauer's blog. Blog, https://phauer.com/2019/modern-best-practices-testing-java/
Haus E (2020) 102. whether or not to repeat yourself: Dry, damp, or wet. Podcast / transcript, https://www.heroku.com/podcasts/codeish/102-whether-to-repeat-yourself-dry-damp-or-wet
Hnat B (2020) Dryer tests - the dumpster fire project. Blog, https://thedumpsterfireproject.com/dryer-tests
Huttunen A (2021) How to make your tests readable - arho huttunen. Blog, https://www.arhohuttunen.com/test-readability/
Jarrett J (2009) Fluent specification extensions - developing on staxmanade. Blog, https://staxmanade.com/2009/02/fluent-specification-extensions/
Johnson K (2008) Is duplicated code more tolerable in unit tests? - stack overflow. Stackoverflow, https://stackoverflow.com/questions/129693/is-duplicated-code-more-tolerable-in-unit-tests
Kainulainen P (2014) Writing clean tests - petri kainulainen. Blog, https://www.petrikainulainen.net/writing-clean-tests/
Kareinen T (2012) Readable tests - tuomas kareinen's blog. Blog, https://tkareine.org/articles/readable-tests.html
Khorikov V (2008) Dry vs damp in unit tests - enterprise craftsmanship. Blog, https://enterprisecraftsmanship.com/posts/dry-damp-unit-tests/
Kochhar PS, Xia X, Lo D (2019) Practitioners' views on good software testing practices. 2019 IEEE/ACM 41st Int. conf. on software engineering, software engineering in practice (ICSE-SEIP), IEEE, pp 61–70
Koskela L (2013) Effective Unit Testing. Manning, https://livebook.manning.com/effective-unit-testing/chapter-4
Lacerda G, Petrillo F, Pimenta M, Guéhéneuc YG (2020) Code smells and refactoring: a tertiary systematic review of challenges and observations. Journal of Systems and Software 167:110610
Lal A (2019) Kotlin dsl - let's express code in "mini-language" - part 5 of 5. Blog, https://www.aditlal.dev/kotlin-dsl-part-5/

Latorre R (2014) A successful application of a test-driven development strategy in the industrial environment. Empirical Software Engineering 19(3):753–773
Lehner D (2021) 3 easy fixes for perfect unit test code - devmate. Blog, https://www.devmate.software/3-easy-fixes-for-perfect-unit-test-code/
Leotta M, Cerioli M, Olianas D, Ricca F (2018a) Fluent vs basic assertions in java: an empirical study. In: 2018 11th International conference on the quality of information and communications technology (QUATIC), pp 184–192, https://doi.org/10.1109/QUATIC.2018.00036
Leotta M, Cerioli M, Olianas D, Ricca F (2018b) Fluent vs basic assertions in java: An empirical study. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp 184–192, https://doi.org/10.1109/QUATIC.2018.00036
Lindner D (2013) unit test - schneide blog. Blog, https://schneide.blog/tag/unit-test/
Lin B, Nagy C, Bavota G, Marcus A, Lanza M (2019) On the quality of identifiers in test code. In: Proceedings - 19th IEEE int. working conf. on source code analysis and manipulation, SCAM 2019, Institute of Electrical and Electronics Engineers Inc., pp 204–215
Lipinski P (2013) or how to write tests so that they serve you well. Presentation, https://2013.jokerconf.com/presentations/03_02_lipinski_pawel_jokerconf-presentation.pdf
Li B, Vendome C, Linares-Vásquez M, Poshyvanyk D (2018) Aiding comprehension of unit test cases and test suites with stereotype-based tagging. In: Proceedings of the 26th conference on program comprehension, p 52–63, https://doi.org/10.1145/3196321.3196339
Li B, Vendome C, Linares-Vasquez M, Poshyvanyk D, Kraft N (2016) Automatically documenting unit test cases. In: Proceedings - 2016 IEEE Int. Conf. on software testing, verification and validation, ICST 2016, Institute of Electrical and Electronics Engineers Inc., pp 341–352
MANJUNATH N, MEULDER OD (2019) No code? no problem - writing tests in plain english - nyt ... Blog, https://open.nytimes.com/no-code-no-problem-writing-tests-in-plain-english-537827eaaa6e
Martin RC (2021) Clean Code: Chapter 9. nan, https://reee3.home.blog/2021/02/17/clean-code-9/
McMinn P, Shahbaz M, Stevenson M (2012) Search-based test input generation for string data types using the results of web queries. In: 2012 IEEE Fifth international conference on software testing, verification and validation, pp 141–150, https://doi.org/10.1109/ICST.2012.94
Meszaros G (2007) xUnit test patterns: Refactoring test code. Pearson Education
Minelli R, Mocci A, Lanza M, (2015) I know what you did last summer-an investigation of how developers spend their time. (2015) IEEE 23rd Int. conf. on program comprehension, IEEE, pp 25–35
Moonen L, van Deursen A, Zaidman A, Bruntink M (2008) On the interplay between software testing and evolution and its effect on program comprehension. In: Software evolution, Springer, pp 173–202
Myers B (2021) Readable test code matters. - brooklin myers. Blog, https://brooklinmyers.medium.com/readable-test-code-matters-e46cc5c411bb
Needham M (2009) Tdd: Test dryness - mark needham. Blog, https://www.markhneedham.com/blog/2009/01/30/tdd-test-dryness/
nickname Company blog F (2016) More readable tests with kotlin - tengio. Blog, https://www.tengio.com/blog/more-readable-tests-with-kotlin/
Oliveira D, Bruno R, Madeiral F, Castor F (2020) Evaluating code readability and legibility: an examination of human-centric studies. In: 2020 IEEE Int. conf. on software maintenance and evolution (ICSME), IEEE, pp 348–359
Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A (2016) Automatic test case generation: What if test code quality matters? In: ISSTA 2016 - Proceedings of the 25th int. symposium on software testing and analysis, Association for Computing Machinery, Inc, pp 130–141
Panichella A, Panichella S, Fraser G, Sawant AA, Hellendoorn VJ (2022) Test smells 20 years later: detectability, validity, and reliability. Empirical Software Engineering 27(7):170
Panichella S, Panichella A, Beller M, Zaidman A, Gall H (2016) The impact of test case summaries on bug fixing performance: An empirical investigation. In: Proceedings - int. conf. on software engineering, IEEE computer society, vol 14-22-May-2016, pp 547–558
Papendieck T (2017) Why sometimes unit tests do more harm than good? Blog, https://www.beyondjava.net/why-sometimes-unit-tests-do-more-harm-than-good
Petersen K, Vakkalanka S, Kuzniarz L (2015) Guidelines for conducting systematic mapping studies in software engineering: an update. Information and Software Technology 64:1–18
Pip C (2020) Clean code in tests: What, why and how? - testproject. Blog, https://blog.testproject.io/2020/04/22/clean-code-in-tests-what-why-and-how/
Posnett D, Hindle A, Devanbu P (2011) A simpler model of software readability. In: Proceedings of the 8th working conf. on mining software repositories, pp 73–82

Ramler R, Klammer C, Buchgeher G, (2018) Applying automated test case generation in industry: a retrospective. (2018) IEEE Int. conf. on software testing, verification and validation workshops (ICSTW), IEEE, pp 364–369

Reagan P (2009) Keep your friends close, but your test data closer - viget. Blog, https://www.viget.com/articles/keep-your-friends-close-but-your-test-data-closer/

Reid J (2016) 3 reasons why it's important to refactor tests - quality coding. Blog, https://qualitycoding.org/why-refactor-tests/

Ricca F, Torchiano M, Di Penta M, Ceccato M, Tonella P (2009) Using acceptance tests as a support for clarifying requirements: a series of experiments. Information and Software Technology 51(2):270–283

Roberts J (2015) Improve test asserts with shouldly - visual studio magazine. Magazine, https://visualstudiomagazine.com/articles/2015/08/01/improve-test-asserts-with-shouldly.aspx

Roberts J (2019) Diagnosing failing tests more easily and improving test ... Blog, http://dontcodetired.com/blog/post/Diagnosing-Failing-Tests-More-Easily-and-Improving-Test-Code-Readability

Rodrigues M (2018) What makes good unit test? readability - matheus rodrigues. Blog, https://matheus.ro/2018/01/15/makes-good-unit-test-readability/

Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In: annual meeting of the Florida Association of Institutional Research, vol 177, p 34

Roy D, Zhang Z, Ma M, Arnaoudova V, Panichella A, Panichella S, Gonzalez D, Mirakhorli M (2020a) Deeptc-enhancer: improving the readability of automatically generated tests. In: Proceedings - 2020 35th IEEE/ACM Int. conf on automated software engineering, ASE 2020, Institute of Electrical and Electronics Engineers Inc., pp 287–298

Roy D, Zhang Z, Ma M, Arnaoudova V, Panichella A, Panichella S, Gonzalez D, Mirakhorli M (2020b) Deeptc-enhancer: Improving the readability of automatically generated tests. In: Proceedings - 2020 35th IEEE/ACM int. conf on automated software engineering, ASE 2020, Institute of Electrical and Electronics Engineers Inc., pp 287–298

Ryswyck JV (2021) Avoid inheritance for test classes - principal it. Blog, https://principal-it.eu/2021/01/avoid-inheritance-for-test-classes/

Sarna A (2018) Do you think your code is perfect? well, think again. Blog, https://blog.knoldus.com/do-you-think-your-code-is-perfect-well-think-again/

Scalabrino S (2019) Automatically assessing and improving code readability and understandability. PhD thesis, Universitá degli Studi del Molise, https://iris.unimol.it/retrieve/handle/11695/90885/92359/Tesi_S_Scalabrino.pdf

Scalabrino S, Bavota G, Vendome C, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2017) Automatically assessing code understandability: how far are we? In: 2017 32nd IEEE/ACM Int. conf. on automated software engineering (ASE), IEEE, pp 417–427

Scalabrino S, Linares-Vasquez M, Poshyvanyk D, Oliveto R (2016) Improving code readability models with textual features. In: 2016 IEEE 24th Int. Conf. on Program Comprehension (ICPC), IEEE, pp 1–10

Schults C (2021) Unit testing best practices: 9 to ensure you do it right. Blog, https://www.testim.io/blog/unit-testing-best-practices/

Setiani N, Ferdiana R, Hartanto R (2020) Test case understandability model. IEEE. Access 8:169036–169046

Setiani N, Ferdiana R, Hartanto R (2021a) Developer's perspectives on unit test cases understandability. In: Proceedings of the IEEE int. conf. on software engineering and service sciences, ICSESS, IEEE computer society, vol 2021-August, pp 251–255

Setiani N, Ferdiana R, Hartanto R (2021b) Developer's perspectives on unit test cases understandability. In: Proceedings of the IEEE Int. conf. on software engineering and service sciences, ICSESS, IEEE Computer Society, vol 2021-August, pp 251–255

Shamshiri S, Rojas JM, Galeotti JP, Walkinshaw N, Fraser G (2018) How do automatically generated unit tests influence software maintenance? In: 2018 IEEE 11th International conference on software testing, verification and validation (ICST), pp 250–261, https://doi.org/10.1109/ICST.2018.00033

Shih J (2020) A field guide to unit testing: readability. Blog, https://codecharms.me/posts/unit-testing-readability

Smart JF (2020) What makes a great test automation framework? - linkedin. Blog, https://www.linkedin.com/pulse/what-makes-great-test-automation-framework-john-ferguson-smart

Snyder D, Kuefler E (2019) Testing on the toilet: tests too dry? make them damp! Blog, https://testing.googleblog.com/2019/12/testing-on-toilet-tests-too-dry-make.html

Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: 2018 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 1–12

testing company (2020) Unit testing tutorial: 5 best practices - vtest blog. Blog, https://www.vtestcorp.com/blog/unit-testing-best-practices/

Tran HKV, Ali NB, Börstler J, Unterkalmsteiner M (2019) Test-case quality-understanding practitioners' perspectives. Int. Springer, Conf. on product-focused software process improvement, pp 37–52

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp 4–15

Vahabzadeh A, Fard AM, Mesbah A (2015) An empirical study of bugs in test code. In: 2015 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 101–110

Van Deursen A, Moonen L, Van Den Bergh A, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001), Citeseer, pp 92–95

Vance S (2013) Quality Code: software testing principles, practices, and ... nan, https://vdoc.pub/documents/quality-code-software-testing-principles-practices-and-patterns-781g7idb34f0

Wagner S, Goeb A, Heinemann L, Kläs M, Lampasona C, Lochmann K, Mayr A, Plösch R, Seidl A, Streit J et al (2015) Operationalised product quality models and assessment: the quamoco approach. Information and Software Technology 62:101–123

Wagner S, Lochmann K, Heinemann L, Kläs M, Trendowicz A, Plösch R, Seidi A, Goeb A, Streit J (2012) The quamoco product quality modelling and assessment approach. In: 2012 34th international conference on software engineering (ICSE), IEEE, pp 1133–1142

Winkler D, Urbanke P, Ramler R (2021) What do we know about readability of test code? - a systematic mapping study. In: Proceedings of the 5th workshop on validation, analysis, and evolution of software tests, in conjunction with the 29th IEEE international conference on software analysis, evolution, and reengineering (SANER)

Winkler D, Urbanke P, Ramler R (2023) Data set for "investigating the readability of test code: combining scientific and practical views". Data Set, https://doi.org/10.48436/w4q8v-28695, https://doi.org/10.48436/w4q8v-28695

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media

Yetistiren B, Ozsoy I, Tuzun E (2022) Assessing the quality of github copilot's code generation. In: Proceedings of the 18th international conference on predictive models and data analytics in software engineering, pp 62–71

Yonekubo T (2021) Readable test code - medium. Blog, https://medium.com/@t-yonekubo/readable-test-code-cad8a7babc7b

Yusifoğlu VG, Amannejad Y, Can AB (2015) Software test-code engineering: a systematic mapping. Information and Software Technology 58:123–147

Zaidman A, Van Rompaey B, van Deursen A, Demeyer S (2011) Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering 16(3):325–364

Zhang B, Hill E, Clause J (2016) Towards automatically generating descriptive names for unit tests. In: ASE 2016 - Proceedings of the 31st IEEE/ACM int. conf. on automated software engineering, Association for Computing Machinery, Inc, pp 625–636

Zilberfeld G (2014) Test attribute #2: Readability - java code geeks. Blog, https://www.javacodegeeks.com/2014/07/test-attribute-2-readability.html

Dietmar Winkler is senor researcher at SBA Research, area manager at the Austrian Center for Digital Production (ACDP), and lecturer at the TU Wien, Vienna, Austria as member of the research group for Quality Software Engineering at the Institute for Information Systems Engineering in the Software Engineering Group. Dietmar holds an PhD and a M.Sc. in Computer Science from TU Wien, Austria. He has more than 20 years of experience in applied research in the fields of software and production systems engineering, engineering process improvement, quality assurance and testing, and empirical software engineering. He (co-)authored more than 130 peer-reviewed publications, (co-)organized international conferences, tracks, and sessions, and served as program committee member for international journals and conferences.

Pirmin Urbanke is a researcher and software engineer at Software Competence Center Hagenberg (SCCH), Austria. He received his B.Sc. and M.Sc. from the TU Wien, Austria. His current research interest is on testing distributed systems.

Rudolf Ramler is a research manager at Software Competence Center Hagenberg (SCCH), Austria. Rudolf holds a M.Sc. in Business Informatics from Johannes Kepler University Linz. He has more than 20 years of experience in applied research in the fields of software engineering, software quality assurance and testing, software analytics, and application lifecycle management. He is author of over 100 reviewed publications, co-organizer and chair of international conferences and workshops, an ISTQB certified tester, and an IEEE and ACM member. His mission and passion are to support industry in turning research results into practically successful solutions.

Authors and Affiliations

Dietmar Winkler[1,2,3] · Pirmin Urbanke[4] · Rudolf Ramler[5]

Pirmin Urbanke
pirmin.urbanke@tuwien.ac.at

Rudolf Ramler
rudolf.ramler@scch.at

1 SBA Research gGmbH, Floragasse 7, Vienna 1040, Austria

2 Austrian Center for Digital Production, Seestadtstrasse 27/19, Vienna 1200, Austria

3 Institute for Information Systems Engineering, TU Wien, Favoritenstr. 9/194, Vienna 1040, Austria

4 Christian Doppler Laboratory on Security and Quality Improvement in the Production Systems Lifecycle (CDL-SQI), TU Wien, Favoritenstrasse 9/194, Vienna 1040, Austria

5 Software Competence Center Hagenberg GmbH, Softwarepark 32a, Hagenberg 4232, Austria