TU WIEN Informatics

A Framework for Evaluating the Readability of Test Code in the Context of Code Maintainability

A Family of Empirical Studies

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Pirmin Urbanke
Matrikelnummer 01527339

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Ass. Dipl.-Ing. Dr.techn. Dietmar Winkler
Mitwirkung: Mag. Rudolf Ramler
Ao.Univ.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Stefan Biffl

Wien, 6. Dezember 2022

Pirmin Urbanke	Dietmar Winkler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

A Framework for Evaluating the Readability of Test Code in the Context of Code Maintainability

A Family of Empirical Studies

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Pirmin Urbanke
Registration Number 01527339

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Ass. Dipl.-Ing. Dr.techn. Dietmar Winkler
Assistance: Mag. Rudolf Ramler
Ao.Univ.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Stefan Biffl

Vienna, 6th December, 2022

Pirmin Urbanke | Dietmar Winkler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Pirmin Urbanke

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

Wien, 6. Dezember 2022

Pirmin Urbanke

Acknowledgements

I thank my supervisor Dietmar Winkler and Rudolf Ramler from Software Competence Center Hagenberg (SCCH) for assisting me in this work with ideas and feedback, which we discussed in many meetings. This assistance also made it possible to publish parts of this work in scientific journals. Of course thanks also go out to family and friends who supported me throughout my study.

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.

Kurzfassung

Kontext und Motivation: Das Testen von Software ist in der Softwareentwicklung eine gängige Tätigkeit mit vielfältigem Nutzen. Es gibt gewisse Garantien, dass das Programm über seinen Lebenszyklus wie erwartet funktioniert, es hilft beim Finden und Ausbessern von fehlerhaften Verhalten, es ist Dokumentation, gibt Verwendungsbeispiele, etc.. Trotzdem wird Testcode oft stiefmütterlich behandelt, was zu Qualitätseinbußen auch in Bezug zur Lesbarkeit führt. Wenn jedoch der Test schlecht lesbar ist, können weiterführende Tätigkeiten wie Wartung von Tests oder das Ziehen von korrekten Schlussfolgerungen basierend auf Tests beeinträchtigt werden. Aber was ist überhaupt lesbarer Testcode? Da Testcode einen anderen Einsatzzweck als Produktivcode hat und über einzigartige Eigenschaften wie Prüfmethoden (assertion methods) besitzt, könnte die Lesbarkeit von Testcode von anderen Einflussfaktoren abhängen als jene von Produktivcode.

Ziel: Wir schlagen ein Framework vor, das zur Bewertung der Lesbarkeit von Testcode verwendet werden kann. Weiters bietet es auch Informationen über Lesbarkeitsfaktoren an, und gibt Best-Practice-Beispiele zur Verbesserung. Neben diesem Hauptziel geben wir einen Überblick über die akademische Literatur auf dem Gebiet der Lesbarkeit von Testcode und vergleichen sie mit den Meinungen von Praktikern. Wir untersuchen die Auswirkung von Codeänderungen (Refactorings), die sich auf die weithin diskutierten Lesbarkeitsfaktoren beziehen, auf die Lesbarkeit von Testfällen. Darüberhinaus sammeln wir Kriterien für die Bewertung der Lesbarkeit aus Freitextantworten, untersuchen den Einfluss der Erfahrung von Entwicklern auf die Bewertung der Lesbarkeit und bewerten die Genauigkeit eines Bewertungsprogramms für die Lesbarkeit, das häufig in anderen Studien verwendet wird.

Methodik: Wir sammeln umfangreiche Informationen über die Lesbarkeit von Testcode, indem wir eine systematische Zuordnung von akademischer Literatur mit den Ergebnissen einer systematischen Zuordnung von Literatur aus der Praxis (sog. grey literature) kombinieren. Wir führen ein Humanexperiment zur Lesbarkeit von Testcode mit 77 erfahrungsmäßig meist angehenden Programmiererninnen und Programmierern aus dem akademischen Umfeld durch, um verschiedene Einflussfaktoren auf die Lesbarkeit zu untersuchen. Wir kategorisieren und gruppieren Freitextantworten der Versuchsteilnehmer und vergleichen die menschlichen Lesbarkeitsbewertungen mit Programmbasierten Lesbarkeitsbewertungen. Schließlich führen wir nach der Erstellung des Frameworks für die

Bewertung der Lesbarkeit, das auf den vorherigen Ergebnissen basiert, eine Evaluierung durch und vergleichen sie mit den Ergebnissen des ursprünglichen Humanexperiments.

Ergebnisse: Die Literaturstudien ergeben 16 relevante Quellen aus der Wissenschaft und 56 Quellen aus der Praxis. Aus beiden Studien geht ein anhaltendes Interesse an der Lesbarkeit von Testcode hervor. Wissenschaftliche Quellen konzentrieren sich auf die Untersuchung von automatisch generiertem Testcode, der oft mit manuell geschriebenen Tests verglichen wird (88%). Zur Erfassung der menschlichen Lesbarkeit werden vor allem Umfragen als Methoden verwendet (44%), die in fast allen Fällen Likert-Skalen enthalten. Bei der praxisbezogener Literatur (56 Quellen) handelt es sich meist um Blogs von Praktikern, die ihre Meinung und Erfahrungen zu Problemen aus ihrer täglichen Arbeit mitteilen. Es gibt klare Überschneidungen bei den Lesbarkeitsfaktoren, die in beiden Gemeinschaften diskutiert werden, aber einige Faktoren sind exklusiv für jede Gemeinschaft. Bei dem Humanexperiment fanden wir einen statistisch signifikanten Einfluss auf die Lesbarkeit von Testfällen bei fünf von zehn untersuchten Codeänderungen, die Lesbarkeitsfaktoren zuzuordnen sind. Wir sehen keinen großen Einfluss der Erfahrung auf die Lesbarkeitsbewertungen, obwohl frühere Untersuchungen einen Einfluss der Erfahrung auf das Verständnis und die Wartungsaufgaben festgestellt haben. Nach der Kategorisierung von rund 2500 Freitextantworten zu urteilen, bewerten die Teilnehmer die Lesbarkeit auf der Grundlage von *Testnamen*, *Struktur* und *Abhängigkeiten* (d. h. testet der Test nur ein Verhalten?). Die Bewertungen des Bewertungprogramms für die Lesbarkeit liegen in etwa 51% der untersuchten Testfälle zwischen dem 0,25% und 0,75%-Quantil unserer menschlichen Bewertungen. Wir haben auch festgestellt, dass unsichtbare Unterschiede in der Formatierung (z. B. Leerzeichen, Tabulatoren) die Bewertungen des Programms um bis zu 0,25 auf einer Skala von 0 bis 1 beeinflussen. Die Bewertung des Frameworks zeigt eine geringere Streuung der Bewertungen zwischen den Teilnehmern und eine höhere Bewertungsgeschwindigkeit im Vergleich zu den Bauchgefühl-Bewertungen der ersten Experimente. Insgesamt bewertet das Framework die Tests zu optimistisch. Allerdings ist die Aussagekraft aufgrund der geringen Anzahl von Umfrageteilnehmern (5) sehr begrenzt. Daher ist diese Evaluierung lediglich ein Konzept, das wir in zukünftigen Arbeiten weiterverfolgen werden.

Conclusio: Bei den Literaturstudien fanden wir unterschiedliche Auffassungen über die Lesbarkeit von Testfällen in der Praxis und in der Wissenschaft, die sich aus den unterschiedlichen Kontexten der jeweiligen Gruppen ergeben. Die Bewertungen des Lesbarkeitsprogramms sind nicht genau genug, um ihnen blind zu vertrauen. Sie müssen noch mit menschlichem Fachwissen ergänzt werden. Unser Framework zur Bewertung der Lesbarkeit ermöglicht eine effizientere Bewertung der Lesbarkeit. Eine groß angelegte Evaluierung ist für zukünftige Arbeiten geplant.

Abstract

Context and Motivation: Software testing is a common practice in software development and serves many functions. It provides certain guarantees that the software works as expected across the life cycle of the system, it helps with finding and fixing erroneous behaviour, it acts as documentation, provides usage examples, etc.. Still, test code is often treated as an orphan, which leads to poor quality tests also with respect to readability. However, if the test has poor readability, upstream activities like maintaining tests or drawing correct conclusions from tests may be compromised. But what is readable test code? Since test code has a different purpose than production code and contains exclusive features like assertion methods, the factors influencing readability may deviate from production code.
Objective: We propose a framework, which can be used to evaluate the readability of test code. It also provides information on factors influencing readability and gives best-practice examples for improvements. Aside from this main goal, we give an overview on academic literature in the field of test code readability and compare it to opinions of practitioners. We investigate the impact of modifications, related to widely discussed readability factors, on the readability of test cases. Furthermore, we gather readability rating criteria from free text answers, investigate impact of developer experience on readability ratings and evaluate the accuracy of a readability rating tool, which is often used in other studies.
Methods: We collect extensive information on test code readability by combining a systematic mapping of academic literature with the results of a systematic mapping of grey literature. We conduct a human-based experiment on test code readability with 77 mostly junior-level participants in academic context, to investigate various influence factors to readability. We categorise and group free text answers from the experiments participants and compare the human readability ratings with tool generated readability ratings. Finally, after the construction of the readability assessment framework, which is based on the previous results, we perform an evaluation and compare it to the results of the initial human-based experiment.
Results: The literature studies result in 16 relevant sources from the scientific community and 56 sources from practitioners. From both literature mappings we see an ongoing interest in test code readability. Scientific sources focus on investigating automatically generated test code, which is often compared to manually written tests (88%). For capturing human readability, they primarily use surveys as methods (44%), which contain

Likert scales in almost all cases. Grey literature (56 sources) mostly consists of blogs from practitioners, sharing their opinion and experience on problems found in their daily work. There is a clear intersection on readability factors discussed in both communities, but some factors are exclusive to each community. For the human-based experiment, we found statistical significant influence on the readability of test cases in five of ten investigated modifications, which map to readability factors. We do not see much influence of experience on readability ratings, although previous research found experience influencing understanding and maintenance tasks. Judging from the categorisation of around 2500 free text answers, the participants rate readability based on *Test naming*, *Structure* and *Dependencies* (i.e., does the test ensure only one behaviour?). The ratings of the readability rating tool are between the 0.25% and 0.75% quantile of our human ratings in around 51% of the investigated test cases. We also found influence of invisible differences in formatting (i.e. spaces, tabulators) affecting the tools ratings up to 0.25 on a scale from 0 to 1. The framework evaluation shows a decreased variation in the ratings across participants and increased rating speed compared to gut feeling ratings from the initial experiments. Overall, the framework rates tests to optimistically. Nevertheless, the validity is very limited, due to a small number of survey participants (5). Therefore, this evaluation is merely a concept, which we pursue in future work.

Conclusion: From the literature mappings we found different views on test case readability between practitioners and academia, which come from the different contexts of the communities. The ratings from the readability tool are not accurate enough in order to trust them blindly. They still need to be complemented with human expertise. Our readability evaluation framework enables a more efficient assessment of readability. A large scale evaluation is planned for future work.

Contents

CHAPTER 1

Introduction

1.1 Context & Motivation

Writing unit tests is a crucial part of software engineering. As software changes over time after the initial development, tests have to be maintained too. It is common knowledge that hard to read production code is hard to maintain. The same also holds for test code, because maintenance tasks usually include reading the code at least once. A lack of readability can lead to developers spending much time on understanding the code and may even lead to the introduction of new bugs, because they misinterpreted some part of the code. Therefore, each person playing a role in the specification, implementation and maintenance of software tests should aim for high readability. The discussion on the readability of (production) code exists for a long time and there exist different approaches to assess its readability. Test code however pursues other goals than production code, hence the aspects which influence its readability may differ. Therefore, **the main goal of this work is to propose a framework for readability evaluation of test code.** In order to reach this goal we gather and compare information on factors influencing readability in academic and grey literature in the course of two systematic mapping studies. We will conduct a readability experiment with humans and an established readability rating tool. From the human-based experiment we extract rating criteria from free text answers and combine them with the other findings to a readability evaluation framework. Finally, we conceptually evaluate the framework with a survey.

The main contributions of this work to scientific community include the individual mapping studies and the comparison of views from academia and practitioners. These provide insights into the relations between these communities and can be used to justify new investigations. The collection of rating criteria and the investigation of influence factors can be used as input for improving the readability of test case generators.
The main contributions for practitioners also include the mapping of both types of literature, because it adds scientific viewpoints on best-practice methods. The readability

assessment framework can be used by testers, developers and instructors as an input during test code reviews, which may speed up the review process and lead to more consistent ratings from all participants. The guidelines give advice on how tests could be improved with respect to readability.

After the overview on this work, we set up the overall context which is software testing in general. Myers et al. [49] give a common definition of testing in *The art of software testing*:
"*Testing is the process of executing a program with the intent of finding errors.*"

Software testing is part of quality assurance, widely adopted by industry (Bertolino et al. [6]) and is a valuable tool to gain confidence in the implemented functionality. However, it not only increases confidence in a product it also secures private data, saves money or lives. A common example for saved money is the short flight and explosion of the Ariane 5 rocket [45], where a 64 bit floating point number was stored into a 16 bit field causing disastrous behaviour. Another example, the so called *Heartbleed* bug [1], found in a popular encryption library affected the whole internet. Being part of many web servers, it allowed attackers to steal communication from 24 - 55% of popular HTTPS sites, Durumeric et al. [18]. This communication could also contain access credentials. What do these examples have in common? They both could have been found with software testing, as probably all faults in software systems.

Besides increasing confidence in the code base, tests can also be seen as documentation. Test cases show developers how the system behaves in normal and exceptional situations. Apart from that, positive test cases act as an example on how the system under test can be used, e.g. they present instantiation of complex objects or basic workflows. To keep time spent with the tests short and in order to allow developers to focus on productive work, test cases should be readable and easily understandable. Testers profit from readable tests, because readable code is usually easier to maintain and to review. Our work supports development of readable tests, because by applying the framework they get trained on spotting test code which could be improved.

Not only testers and developers profit from readable test code but also test case designers might find it valuable to know, if different kinds of tests deviate strongly in readability. By combining the readability assessments of test cases from classes and modules, managers and team leaders get an overview on a additional quality aspect of their test suite, which may give hints to hard-to-test production code.

Although it is possible to write unit tests with standard libraries from programming languages, a dedicated testing framework provides useful functionality for setting up and executing tests. The most important aspect are the so-called `assertions`, which come in different flavours. Assertion methods allow the programmer to check that the program is in a certain state at a specific point in the execution. When the program satisfies the state, the test continues, else it fails with a message to the programmer.

[1] CVE-2014-0160 The Heartbleed bug https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160, visited: 2021-06-12

1.2 Problem Description

Readability is a quality criteria which influences maintainability, because a developer might have to adapt tests written by someone else. At some point the developer might have to decide if a test case should only be modified or if a complete rewrite would improve future maintenance tasks. In this case a framework for assessing readability can deliver valuable insights, which can also be reproduced by other developers. Also no matter how the developer decides, the developer has to know how the test case should be modified, in order to improve the readability.

Software tests can be used by developers as additional documentation of the functionality of certain parts of the system. Besides, developers also have to write tests themselves when doing test driven development or have to review test code, so they clearly profit from a framework for readability assessment we propose. Test case designers might also find the results of the work useful, because by knowing which tests are rated as more readable, they can adapt their design to improve future tests. Team leaders and managers get an overview on a quality aspect of their test suite which can also be a hint to problems in production code. Therefore, the main goal of the work is to propose a framework for readability assessment of test code.

When it comes to test code quality, the term *test smell* will pop up sooner or later. Test smells are a special version of code smells only occurring in test code. This term was coined by Van Deursen et al. [70]. Code smells are not directly bugs in the code but more like unfortunate code design decisions, which can lead to problems in the future. The term *code smell* was invented by Kent Beck and got the attention of a larger audience in a book by Martin Fowler [23], who defines code smells as: '*A code smell is a surface indication that usually corresponds to a deeper problem in the system*'.[2] This definition is already a hint, that test smells capture a different problem and do not necessarily have to do with test code readability.

As an example for the problem with test smells, the following Listing 1 shows a test with two smells. In the test case we first initialize an object of the imaginary `Calculator` class. Then we test this object by using assertion methods provided by the testing framework. The `assertEquals()` method assures that the first parameter (the expected parameter / ground truth) is equal to the second parameter. We assume that the methods `add()` and `multiply()` return an integer. The third parameter of the assertion is unused. All assertions of the JUnit framework have this optional parameter, which can be used to define a custom message printed to console in case this assertion fails. Coming to the actual smells, the listing firstly contains an *Assertion Roulette* smell, because there are multiple assert statements with the message parameter unused. This can make the failing assertion harder to find. Secondly, the test has a *Eager Test* smell, because different methods of the system under test are tested in a single test case (methods `add` and

[2] Martin Fowler, blog entry: CodeSmell https://martinfowler.com/bliki/CodeSmell.html visited: 2022-11-21

`multiply`). Thirdly the *Magic Numbers* smell is present, because integer literals instead of variables or constants with a descriptive name are used.

```
1 // SMELLY TEST CASE
2
3 @Test
4 public void testAddAndMult() {
5   Calculator calc = new Calculator();
6   // Test smells: Eager Test, Assertion Roulette, Magic Numbers
7   assertEquals(5, calc.add(2, 3));
8   assertEquals(4, calc.multiply(2, 2));
9   }
```

Listing 1: Test smells in this listing lines 7-8: Eager test: Different methods are tested in one test method. Assertion Roulette: the message parameter of the assert statements should be used. Magic Numbers: Descriptive variables or constants should be used instead of integer literals. Refactored version in Listing 2.

A refactored version of this test, where these three smells are removed, could look something like Listing 2. By giving each assertion its own test method, only one thing is tested at a time. Since the tests only contain one assertion, the message parameter can be left unused. The setup for the tests is the same, therefore it is extracted into a separate `@BeforeEach` method, which gets executed before each test method. The integer literals are replaced with local variables. However, is the refactored version really better in terms of readability or would a less aggressive refactoring be more readable? Consider that, for example, the refactored version uses much more lines, methods and variables than the smelly version.

Like natural language texts, source code too can be more or less readable. Therefore, code readability is determined by how easily humans can understand a given piece of source code. Of course this feature is subjective but there still are traits in code which support or prevent readability. Buse et al. [8], Posnett et al. [57] or more recently Scalabrino et al. [64] investigated such human preferences and constructed a readability metric for source code. Scalabrino et al. [64] published a tool, which generates readability ratings for Java source code. While this tool allows to rate readability of large amounts of code, it is trained for all kinds of code and not specifically for tests. Since test code has to achieve different goals than production code, the tools response may be a bit fuzzy.

The rest of the work is organized as follows. In the next chapter, Chapter 2, we provide an overview on the related work concerning test code quality and readability. Next, in Chapter 3 we present and justify our research questions, which we use to gather information for the proposed readability framework. The questions are grouped by the methods utilised and answered in order of appearance, hence Chapter 4 deals with the systematic mapping study in academic literature, Chapter 5 answers questions related

```java
// REFACTORED TEST CASE

private Calculator calc;
@BeforeEach
public void setUp() {
  calc = new Calculator();
}

@Test
public void testAdd() {
  int result = 5;
  int addend1 = 2;
  int addend2 = 3;
  assertEquals(result, calc.add(addend1, addend2));
}

@Test
public void testMultiply() {
  int result = 4;
  int multiplicand = 2;
  assertEquals(result, calc.multiply(multiplicand, multiplicand));
}
```

Listing 2: JUnit5 tests with a setup method which is called before each test method. The tests themselves only assert one thing at a time. Integer literals in the assertion are replaced with variables. Original version in Listing 1.

to results of the grey literature study and Chapter 6 reports results of an experiment concerning test code readability in academic context. In Chapter 7 we take key findings of the previous chapters into account to present a proposal for a readability framework, which we conceptually evaluate with a survey. We discuss the result and list threats to validity of this work in Chapter 8. Finally, we conclude with Chapter 9 containing a summary of the findings and prospect on future work.

CHAPTER 2

Related Work

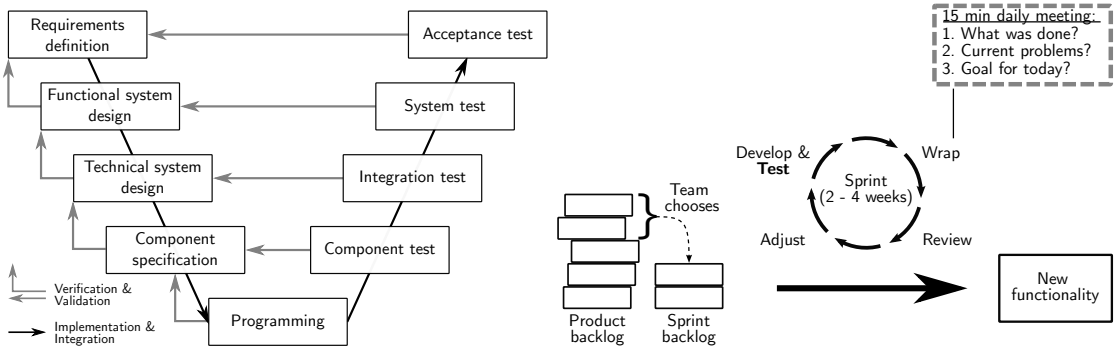
After the general introduction to the context and the problem at hand, we firstly take a look at the test process, which ensures a well structure approach and test code generators, which can generate test suites for large programs in short time. Then we provide a short overview on (test) code readability and the topic of test smells, which are also relevant to test code quality.

2.1 Test Process

Software systems are tested on different levels and also in different stages of development. Figure 2.1 shows two common representations of software development processes. Figure 2.1a shows the V-model by Spillner et al. [68], it presents the different steps in software development with their corresponding testing counterparts. Unit testing or component testing is situated on the lowest level and is concerned with the functionality of one small part of a program. In object oriented programming often a class and its methods are unit tested. The next level is integration testing, where the interaction of these units is investigated. Next the tests on system level are performed, which ensure the functionality of the complete system. The last test level, the acceptance test assures the conformance to the customers requirements. More detailed descriptions of this and other levels of testing, which are utilised depending on the requirements are omitted, because in this work the focus lies on unit tests.

A more modern process of software development shown in Figure 2.1b is the SCRUM process, which is a common proxy for agile project management. The members of a SCRUM team choose their tasks for a development period, the *sprint*, from a product backlog themselves. The period of a sprint usually lasts for two to four weeks. At the end of a sprint the new functionality is presented to the customer. In contrast to the V-model tests for a functionality are written during development, which again highlights the importance that developers should also know how to write good tests.

Although it is not exclusive to the SCRUM model, **t**est **d**riven **d**evelopment (TDD) as proposed by Kent Beck is closely connected to agile software development. According to the preface of Kent Becks book on TDD [4] it follows two simple rules which are "*Write new code only if an automated test has failed*" and "*Eliminate duplication*". From these rules he derives the well known development cycle: *Red* → *Green* → *Refactor*. In the first step 'red' the desired behaviour of the actual program is written in a test case. The test case does not have to compile at this stage. Next, in the 'green' stage, the program is implemented in a fast, maybe unclean way, until the test passes. Finally, in 'refactor', the previously fast written code is cleaned up. In TDD every development starts with a test case and the test cases determine the behaviour of the program. Hence readable tests allow for faster development, because developers can identify the requirements of the test case more easily.



(a) The traditional V-model shows different steps in software development with their corresponding testing counterparts. Figure redrawn; original from Spillner et al. [68].

(b) Agile software development process in the SCRUM model. Testing is done in each sprint.

Figure 2.1: Common representatives of traditional and agile software development processes.

The tests themselves are executed in different environments, for example on the developers machine (Beller et al. [5]) or automatically during the build process when continuous integration is used (Fowler [19], Meyer [48]). Because programming languages like Java, C# or C++ do not explicitly provide testing facilities in their standard libraries, testing frameworks like JUnit (Gamma et al. [26]) for Java were developed.

But how do the developers or testers know when to stop testing, because they tested the complete system? This is where test metrics come in to play. As an example the books by Jorgensen et al. [35] or Kaner et al. [36] list several metrics. A fairly simple one is *code coverage*, where the lines of code, which are reached by the test cases are counted. A little bit more evolved is the *branch coverage*, where the execution paths are counted. E.g. an if-then-else structure has two branches, the `if` and the `else` branch. This metric is useful for finding missed execution paths fast. With additional tools e.g.

Apache Maven for Java and appropriate plugins, such coverage criteria can easily be setup as quality criteria which give developers instant feedback after each execution of the test suite. To fulfil such coverage metrics without much effort, test case generation suites like Randoop [51] or Evosuite [25] exist.

2.2 Test Case Generators

Although test case generators like Randoop [51] or Evosuite [25] can quickly generate large test suites, they both produce badly readable or maintainable tests, see Listing 3 and Listing 4 for examples from both generators. Both samples look rather chaotic, their test naming just consists of 'test' and an consecutive number, the variable names consist of class names and again consecutive numbers. In the Evosuite test in Listing 3 on the plus side there is some structure indicated with the empty lines. However it is used for separating not only different behaviours from one class but also from different ones (i.e. `StringUtils` and `Stack`). From this perspective the name of the test itself comes to no surprise, because finding a reasonable name for such a test is at least challenging. In the Randoop generated test in Listing 4 there are no empty lines for structuring but all assertions are grouped at the bottom of the test, which is a common best-practice because it suggests a '*Arrange Act Assert*' structure. However, this is not the case here, because two *Arrange* and *Act* steps are performed consecutively. The first two steps go from line 8 to 12, the second steps from 13 to 16. Finding an appropriate test name is also challenging, even if all tested methods come from the `StopWatch` class. Last but not least the block of assertions at the bottom looks intimidating but effectively they only check three `booleans` which should be false, one `string` representation (line 18) and equalness of a `long` value (line 21). The first parameter of the assertions is used for constructing an assertion message, which could be avoided by using the appropriate assertion, which generate these messages automatically. Besides it would also get rid of comparisons like `boolean 4 == false`.

As we have seen there is much potential for improvements in test case generation. Therefore, researchers propose various enhancements for this tools. Palomba et al.[53] observe that generated tests with low cohesion (one test case should only test few behaviours) and high coupling (multiple tests test the same behaviour) negatively impact maintenance activities. Therefore, they include this criteria in the generation algorithm and show in an empirical study that their tests had less coupling and higher cohesion. Daka et al.[14] include readability criteria based on a human model into the generation algorithm of Evosuite and demonstrate this approach on selected Java classes. The improved Evosuite algorithm produces slightly more readable tests without loss of code coverage. Robinson et al. [58] present some techniques for generating more maintainable regression unit test suites. In experiments with an industrial system their tests needed fewer edits when the system under testevolved and their tests were considered readable by the developers of the system.

Roy et al. [60] improve readability of tests by generating test cases summaries and

```java
@Test
@Timeout(4000)
public void test303() throws Throwable {
  boolean boolean0 = StringUtils.isAlpha((CharSequence) null);
  assertFalse(boolean0);

  String[] stringArray0 = StringUtils.split(", '");
  assertNotNull(stringArray0);

  Stack<Object> stack0 = new Stack<Object>();
  assertEquals(0, stack0.size());
  assertTrue(stack0.isEmpty());
  assertTrue(stack0.empty());
  assertEquals("[]", stack0.toString());
  assertEquals(10, stack0.capacity());
  assertNotNull(stack0);

  // Undeclared exception!
  // Index: 2791
  Exception e = assertThrows(IndexOutOfBoundsException.class,
                             () -> stack0.listIterator(2791));
  verifyException("java.util.Vector", e);
}
```

Listing 3: Test case generated with Evosuite

renaming identifiers and test names with a deep learning model. They integrate this tool into EvoSuite and show with a survey that this approach increases readability.

Daka et al. [12] propose a model for classifying the readability of unit tests. They also integrated it into Evosuite and evaluated the effectiveness of their model with an experiment. Participants preferred the improved tests, although answers to questions on the test code were as accurate as for the unimproved version. The code factors influencing readability identified by Daka et al. are used by Setiani et al. [65] in combination with developer related metrics for a new readability model.

2.3 (Test) Code Readability

Since the advent of programming languages the readability of the code written in these languages has been of relevant concern. COBOL is probably the first language which tried to achieve high readability with sticking to English language as close as possible [61]. This is in contrast to a nearly equally old language like ALGOL which appears more like a language like C or Java. But just because COBOL is considered to be a language with many unreadable programs, this does not mean that English-like syntax in programming languages is bad. Consider SQL for example, which contains a magnitude of optional

```java
public static boolean debug = false;

@Test
public void test551() throws Throwable {
  if (debug)
      System.out.format("%n%s%n", "RegressionTest1.test551");
  StopWatch stopWatch1 = new StopWatch("");
  stopWatch1.reset();
  stopWatch1.reset();
  boolean boolean4 = stopWatch1.isSuspended();
  java.lang.String str5 = stopWatch1.toString();
  boolean boolean6 = stopWatch1.isSuspended();
  stopWatch1.start();
  stopWatch1.stop();
  boolean boolean9 = stopWatch1.isStarted();
  long long10 = stopWatch1.getStartTime();
  assertTrue("'" + boolean4 + "' != '" + false + "'", boolean4 == false);
  assertEquals("'" + str5 + "' != '" + "0:00:00" + "'", str5, "0:00:00");
  assertTrue("'" + boolean6 + "' != '" + false + "'", boolean6 == false);
  assertTrue("'" + boolean9 + "' != '" + false + "'", boolean9 == false);
  assertTrue("'" + long10 + "' != '" + 1592683830012L + "'",
             long10 == 1592683830012L);  // flaky
}
```

Listing 4: Test case generated with Randoop

keywords, which make queries appear more like English sentences than program code. And even more recently a trend towards *Behaviour Driven Development* with frameworks like Cucumber and the language Gherkin[1] enforce the goal to write automated tests in natural language, see Listing 5.

```
Feature: Checkout

  Scenario: Pay order with filled cart
    Given the Customer has filled shopping cart
    When the Customer clicks the 'Order now' button
    Then the Customer must have enough money on their payment method
```

Listing 5: Test case written in the language Gherkin

After this short introduction on code readability we turn the focus on recent research on this topic. Grano et al. [30] conclude that manually written test code is less readable than production code, but automatically generated tests are even less readable. Lin et al. [43] come to the same conclusion for identifiers and also list characteristics for different

[1] Gherkin reference: https://cucumber.io/docs/gherkin/reference/

qualities of identifiers. However, Grano et al. [30] used a model to compute readability, which could give different ratings than humans.

Concerning code readability metrics Buse et al.[8] constructed a general code readability metric using structural properties of code like line length, amount of keywords or length of identifiers. With their metric they show that a selected set of large software projects tend to get more readable as they get more mature. Choi et al. [11] use similar metrics in their approach. Xu et al. [75] compute readability with word correctness and the property of an identifier being memorable. A more recent general readability model was proposed by Scalabrino et al. [64] who combine structural and textual features like readability of comments or consistency of identifiers, which were not considered by previous work in this breadth. However, we do not know how well these models perform when rating test code.

Oliveira et al. [50] provide an overview on general code readability and propose an informal definition of readability and legibility because these and similar terms are often used interchangeably in the field of software engineering. In this work the focus lies on specifically on test code, which is different to production code.

Although Scalabrino et al. [63] indicate that there is no correlation between automatically generated readability ratings and understandability of source code, we still assume that most programmers and testers find more joy in working with readable test code (apart from a masochistic point of view).

2.4 Test Smells

When researching test code quality test smells are a prominent topic. Van Deursen et al. [70], define a set of eleven test smells which are regularly referenced by publications on this topic. The authors also provide suggestions on how these smelly tests can be refactored.

Ceccato et al. [9, 10] and Shamshiri et al. [67] performed experiments in the field of test smells and maintenance of test code. The accuracy of performing maintenance tasks like bug-fixing, is equal, no matter if a test was generated or written by humans. Also Ceccato et al. state that developers experience plays an important role when working on such tasks. However, the relation between experience and perceived readability of test cases is still an open question.

Test smells are highly diffused in test code and they also appear in other languages than Java, which is focused by researchers. De Blesser et al. [15] investigated diffusion of test smells in Scala projects. Palomba et al. [52] and Grano et al. [31] report that test case generators like Randoop, Evosuite or JTExpert often add high amounts of specific smells to their tests.

In the domain of test smells Garousi et al. [28], [27] conducted a large multivocal literature mapping, which differs from a standard SLR in the way, that *grey literature* for

example blogs, forums and videos is included alongside scientific sources, after a rigorous examination of the sources quality. With this approach the authors gather 166 sources and extract 196 names for test smells. These are an excellent source for classifying unknown smells, although, due to the sheer amount of smell names, some are synonyms for the same smell.

For searching test smells in large amounts of source code, tool support is available. Bavota et al. [3] proposed one for Java, which was also adopted by other researchers. However, also other tools and metrics are available for various languages. Khom et al. [39] propose a Bayesian approach for detecting smells, De Bleser et al. [16] propose a smell detector for Scala. Fernandes et al. [21] performed a systematic literature review and provide a list of available tools. They found 61 tools, where 29 were available for download and covered different languages.

Our work will sketch the landscape of test code readability research with a systematic mapping study, which is a more specific topic than the overview for general code readability by Oliveira et al. [50]. We further enrich the literature study by including grey literature from practitioners to investigate which influence factors to readability are discussed in practice. With an initial readability study with students we investigate the influence of a selected set of factors on readability. Furthermore we extract readability criteria from free text rating explanations gathered in the experiment, because broad investigations on such factors are hardly available in the context of test code readability. The work by Setiani et al. [66] investigates such criteria but differs in the selection of the test cases, because they use constructed examples and we use test cases from open source software projects. We analyse the accuracy of a readability rating tool, because the tool may not be specialised on test code and might give different ratings than humans. Finally, we investigate the impact of developer experience on readability ratings.

CHAPTER 3

Research Questions

In order to propose a framework for readability assessment of test code, we need broad knowledge on test code readability. We start with a short explanation on the research approach, which is the theoretical foundation of this work. Afterwards we present the concrete research overview with an explanation of the individual steps of this work. Finally, we define the research questions we derived to design a framework for readability assessment of test code.

3.1 Design Science Cycle

We follow a process inspired by **Design Science Frameworks** proposed by Hevner et al. [33] and Wieringa [71]. Figure 3.1 shows how this work fits into the design science framework by Wieringa.

Since the sentence '*L'art pour l'art*', does not apply to science, there exists a *Social context* in every research project (see top of Figure 3.1). It is defined by the stakeholders of the project and amongst others contains possible users like testers; developers or instructors, or profiteers like the scientific community. The social context defines the goal of this work to *Propose a framework for readability assessment of test code*, because as stated in Section 1.2, test code readability is a relevant problem in software engineering.

Wieringa summarises design science as '*Design science is the design and investigation of artifacts in context.*'. The artifacts under investigation target to solve a problem in the real world. These problems are called *Design problems* (see the box 'Design' in the middle left of Figure 3.1). Potential solutions to design problems can have many forms, some of which manage to solve the problem completely or partially, while others completely fail to solve the problem or instead they may solve another design problem. In this work the design problem is the main goal of this work, which is to *Propose a framework for readability assessment of test code.*

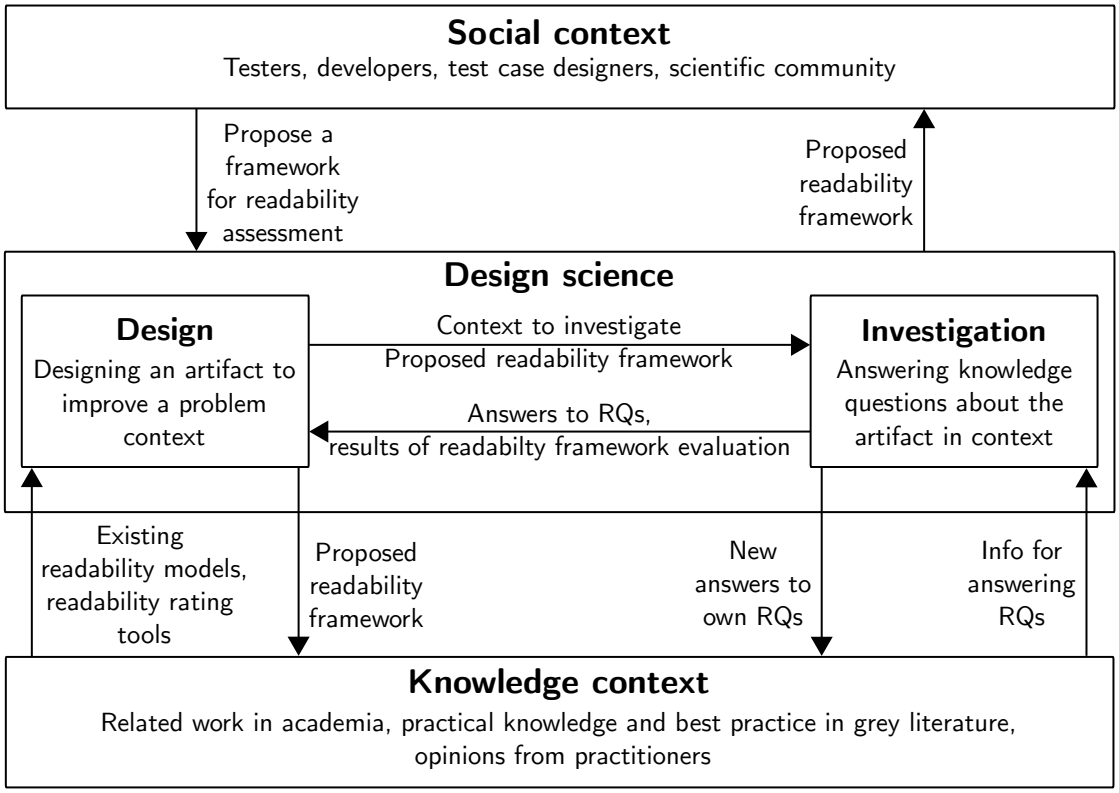


Figure 3.1: Redrawn design science framework from Wieringa [71], adapted to the research approach of this work.

In order to increase the odds of designing an artifact, which solves or improves the posed problem, we investigate the context in a first step (see the left to right arrow in the center of Figure 3.1). This investigation answers so-called *Knowledge questions*, which are not only related to the artifact in the context but also the context itself. The knowledge questions in this work are primarily defined by the research questions in this chapter.

We answer the research questions with information from the *Knowledge context* (see bottom of Figure 3.1), which contains related work in academia, knowledge from practitioners and other sources. By publishing this work we also give back information to the knowledge context with the answers to the research questions and finally with the proposed readability framework.

When we answered the research questions (see the right to left arrow in the center of Figure 3.1), we end the first iteration of the design science cycle and design the artifact, which is the readability assessment framework. Existing readability models or rating tools from the knowledge context also influence the design of the artifact.

In the second and last iteration of the design science cycle in this work we evaluate the readability framework in the context. This will show strengths and weaknesses of the

proposed solution. The insights from this evaluation could then be used to improve the design.

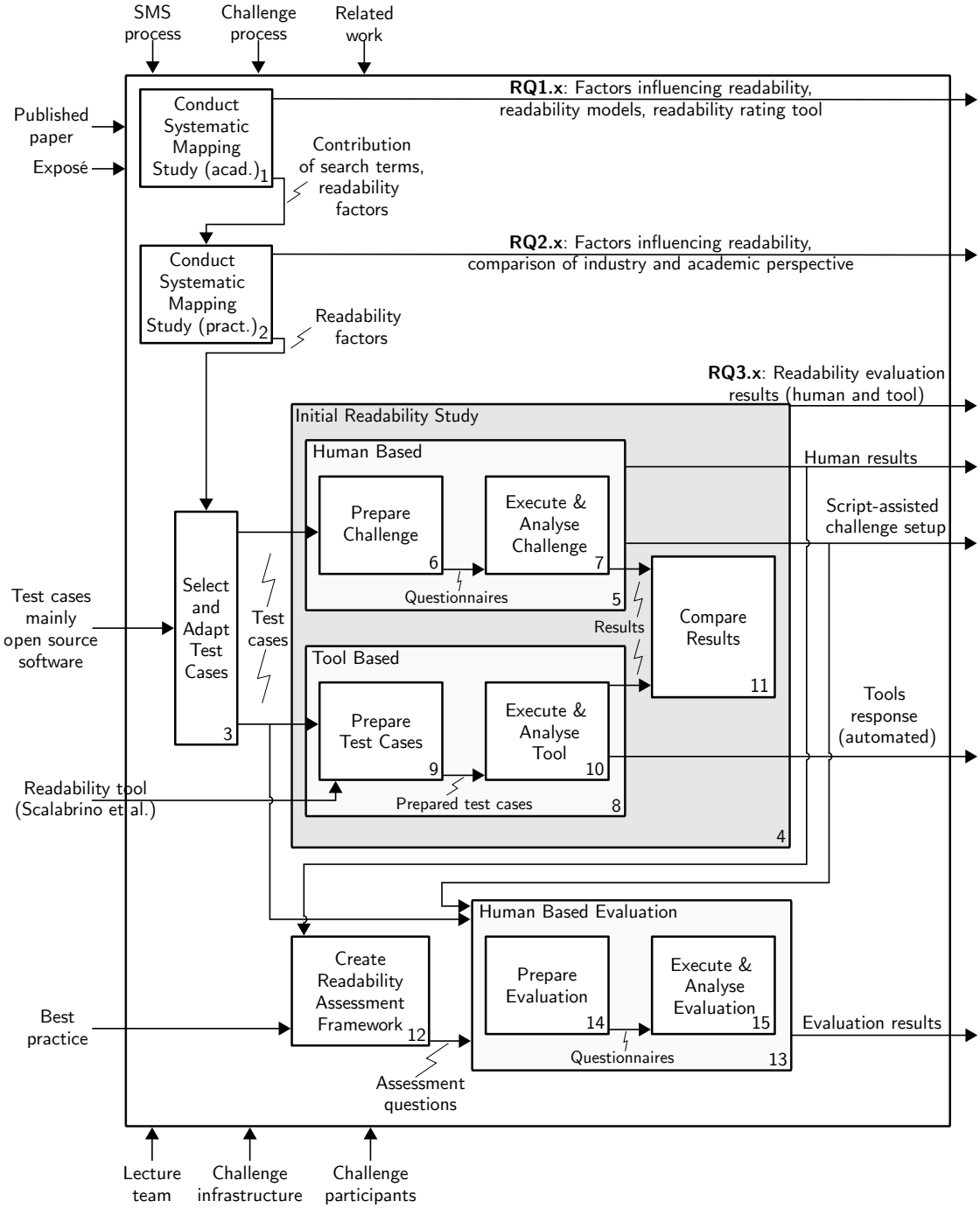


Figure 3.2: Research overview in IDEF0 notation.

3.2 Research Overview and Research Questions

We provide an overview on the concrete research of this work followed by a reasoned definition of research questions and applied methodology. Figure 3.2 shows an overview of the research process in IDEF0 notation[1]. IDEF0 diagrams use the ICOM semantic, which assigns objects, which take part in an activity or function (the rectangles) a special role in this activity, depending on which side of the rectangle the object enters the activity. ICOM stands for:

Input: the left edge of activities are inputs which will be transformed into the output. E.g.: the complete research process uses the inputs published paper, exposé, etc.

Control: the top edge of activities is meant for objects which ensure that the activity produces a correct output.

Output: the right edge of activities is reserved for outgoing arrows, the results of the activity.

Mechanism: the bottom edge of activities describes what the activity needs to transform input to output. This can be tools, infrastructure, persons etc.

One notable overall input for this work is the published paper (Winkler et al. [72]) and another paper, which is currently under review for publication (Winkler et al. [73]). The author of this work conducted the research presented in these papers under supervision of Dietmar Winkler and Rudolf Ramler, which are also main advisor and main assistant of this work. The text passages, figures and tables, which were taken from these papers were created by the author of this work. At the start of each relevant chapter a note clarifies the origin of these passages.

We start this work with a systematic mapping study (*SMS*) on academic literature (see activity one in the top left in Figure 3.2). With this we obtain answers related to RQ1 and its sub-questions. That is, we get an overview of the publication landscape, the kind of methods used, the types of conducted studies, factors influencing readability, readability models and a readability rating tool. The process of the SMS assures a certain reproducibility of the results.

The contribution of search terms i.e. 'Which search terms lead to relevant search results?', and the set of readability factors act as control for the second activity, a SMS on so-called grey literature. According to Garousi et al. [29], who compose guidelines for grey literature reviews in software engineering, a widely accepted definition is '*<grey literature> is produced on all levels of government, academics, business and industry in print and electronic formats, but which is not controlled by commercial publishers, i.e.,*

[1] ISO/IEC/IEEE 31320-1:2012(en) Information technology — Modeling Languages — Part 1: Syntax and Semantics for IDEF0. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:31320:-1:ed-1:v1:en, accessed: 2022.11.23

where publishing is not the primary activity of the producing body' by Farace et al. [20]. The contribution of search terms guide the search terms of the grey literature search to obtain accurate search results. We map the found influence factors in grey literature to the ones found in academic literature and compare the perspectives from industry and academia. This results in answers related to RQ2.x.

The combined readability factors from step two control the selection of test cases in step three. The test cases mainly originate from open source software projects, a small proportion comes from student-written test cases. We create alternative versions of the test cases with respect to the readability factors with best practice approaches.

We use the adapted test cases in our initial readability study (see activity four), which consists of two different kinds of evaluations. On the one hand we have a human-based experiment in the form of a testing challenge, where we first create surveys with *Google.Forms*. These contain screenshots of test case snippets, which have to be rated and reasoned concerning the readability of the tests. The challenge participants are students from different runs of a software testing course at TU Wien. On the other hand, in the tool-based experiment we use the readability rating tool from Scalabrino et al. [64] to obtain readability ratings for the same test cases as in the human experiment. In order to get consistent ratings from the tool we first prepare the test cases by unifying certain aspects of formatting. When both experiments are finished, we compare the tools result to the readability ratings of the participants. Summarising, with the initial readability study we answer questions related to RQ3.x, obtain analysed results for the human and the tool rating and a script-assisted challenge setup for creating similar surveys in the future.

Armed with the human results, which contain readability rating criteria from the participants and factors influencing readability, we create the readability assessment framework in activity twelve. The framework is based on best practice examples and contains a set of assessment questions which target factors influencing readability.

In the last steps we evaluate the readability framework with a survey with a selected set of test cases. We create the survey with *Google.Forms* assisted by the previously created script. Finally, after the execution of the survey the analysis of the answers shows the evaluation result, which concludes the research overview.

RQ1: What do we know about test code readability in academic literature?

The first question is refined and split up into the following sub-questions. It lays the foundation for this work and we will answer this questions with a SMS on academic literature. In the context of literature studies in research projects there also exists another approach, the systematic literature review (*SLR*).

SLRs as proposed by Kitchenham et al. [40] are to some extent similar to a SMS but their goals and therefore some of their methods differ from each other. A SLR has

clearly defined goals and aggregates and analyses primary studies to gather evidence on effectiveness or usefulness of a process. This requires an in-depth analysis of the quality of the studies, which can limit the amount of studies under investigation.

In contrast, according to Petersen et al. [56] a SMS is more explorative than a SLR, since it aims to give an overview on the publication landscape of a given field. This can be used to discover research trends or gaps. Research questions can be formulated more generic and a rigorous quality control of the selected studies is not mandatory, therefore larger quantities of studies can be processed. A key part is the mapping or clustering which summarises aspects of the studies of the field.

We use the SMS approach, because we want to obtain an overview of test code readability in scientific literature and because our goal is not to investigate the effectiveness or usefulness of a specific process in this stage of the work. Related to this part of the work is Oliveira et al. [50] who give an overview on general code readability, whereas we specifically focus on test code readability.

RQ1.1: What is the importance of test code readability in scientific communities?
With this RQ we want to obtain an overview on the demographic of publications concerning timeline and venues.

RQ1.2: Which types of studies are published and which research methods are used?
This RQ provides an overview on the types of studies and the methodology used by these studies. The results of this question will provide a benefit to our own selection of methods.

RQ1.3: Which factors influence readability of test code?
Code can vary in many ways and we are interested which e.g. structural features like line length have a relevant influence on readability.

RQ1.4: Which kinds of tests were investigated?
Software tests can come in various flavours ranging from unit tests to automated GUI tests or even test scripts for manual execution. Depending on the kind of test different readability criteria will be relevant.

RQ1.5: Which executable models for assessing code readability exist?
For constructing our framework we would like to have some tool support. Therefore we will systematically search for tools in academic literature starting from the raw search results of the SMS.

RQ2: What is the opinion of practitioners on test code readability?

Software testing is a common practice in software development, therefore we enhance our SMS of academic literature with views from practitioners with a SMS of grey literature.

According to guidelines for including grey literature in research proposed by Garousi et al. [29], this work can be classified as a multivocal literature mapping. For this part of the work we follow a similar process like in the previous SMS for academic literature, with the guidelines from Garousi et al.[29] in mind.

RQ2.1: Which influence factors are discussed in grey literature?
Instead of asking a selection of practitioners directly, we search the internet for sources, which discuss test code readability and extract the influence factors. We use this approach, because we assume that many developers use the internet as a source of information and inspiration.

RQ2.2: What is the difference between influence factors in scientific literature and grey literature?
The focus of scientists and practitioners on influence factors might be different. A comparison of the viewpoints will broaden our understanding of the topic.

RQ3: What insights can we obtain from a readability experiment with students?

We conduct a human-based readability experiment with students over multiple years. The participants come from software testing courses at TU Wien and were asked to rate code snippets and explain their rating with free text answers in an online setting. We use an online setting, because it is more convenient for the participants which we hope positively affects the amount of participants. The disadvantage of this approach is the decreased the level of control over the experiment participants, e.g. the participants can be disturbed by random events or they could affect their ratings by talking to each other, etc..

In order to increase the practical relevance of the experiment we mostly use test cases from open source software projects. This avoids to rely on toy examples and some bias in the construction of the tests, because the tests were not written by the author or someone related to the author of this work. The selection of test cases is based on the readability factors found by the previous research questions. The experiment follows an A/B testing approach, where some participants evaluate the readability of a unmodified version of the test case and other participants do the same for a modified version.

We conduct a technology-based readability experiment with the tool from Scalabrino et al. [64], with the same test cases as the human-based experiment. This allows us to compare both ratings.

RQ3.1: Do factors discussed in practice show an influence on readability when scientific methods are used?
We study the impact of a selection of readability factors by analyzing results of the A/B testing experiment. We perform the statistical analysis in R on level a of

$\alpha = 0.05\%$ starting with a Shapiro-Wilk test to check for normal distribution of the data. Normally distributed data would allow us to use a parametric test like the t-test for unpaired data, which has more statistical power than the non-parametric Wilcoxon Rank Sum test. We test on the difference of ratings in the A/B groups and report the effect size with Cliff's Delta (δ). We interpret the returned effect size according to Romano et al. [59] with $|\delta| < 0.147$ *"negligible"*, $|\delta| < 0.33$ *"small"*, $|\delta| < 0.474$ *"medium"*, otherwise *"large"*. This approach is also used by [60] for their Likert scale data.

RQ3.2: What criteria do students use for readability ratings?
Human readability judgement is a complex process. We evaluate and categorise free text answers of the experiments participants. We hope that we can synthesize some commonalities which will enhance our framework. Setiani et al. [66] ask a similar question however, our experiment differs in test cases and participants.

RQ3.3: What is the impact of developer experience in context of readability?
While the impact of developers experience on maintenance activities like debugging was investigated by Ceccato et al. [9, 10] this question was not asked directly. We expect similar results, because performing maintenance tasks with software tests usually includes reading the test case at least once.

RQ3.4: What is the accuracy of automatic readability assessment in comparison to students rating?
In order to decide if the tool from Scalabrino et al. [64] found in RQ1.4 is useful for us, we evaluate if its ratings are comparable to human ratings.

CHAPTER 4

Systematic Mapping Study (SMS)

This part of the work focuses on the Systematic Mapping Study (*SMS*). After a coarse description of the process we cover each step in detail. Finally, we show the results and give answers to RQ1 and its sub-questions.

Note: Main parts of this chapter are already published as Winkler et al. [72] and under review for another publication as Winkler et al. [73]. The parts concerning the contribution of search terms (Subsection 4.2.2) and the search for a readability tool (Subsection 4.2.7) are novel in this work.

4.1 SMS Protocol & Process

Systematic mapping studies are a great opportunity for preparing new primary studies as described by Petersen et al. [55, 56]. The major difference between an ordinary literature review and a SMS is the systematic approach, which allows other researchers to reproduce the results of the mapping. To accomplish this, all activities during the recherche, like the used search engine, search string and initial results are rigorously documented.

Figure 4.1 presents a coarse representation of the procedure. In the first two steps we define a search string, which we execute on search-able databases for scientific, peer-reviewed literature, which are Scopus, ACM and IEEE. After deduplicating the results of the queries, we sift out studies irrelevant for this topic based on a set of in- and exclusion criteria by title and abstract of the study. If the abstract and the title do not provide enough information, a full text reading is performed. In step three we perform backward snowballing with the intermediary result in order to find important literature not already included in the set. With this step finished, we obtain our final result in step four after filtering and deduplicating the results from the backward snowballing. We use this result to answer RQ1.1 - RQ1.4.

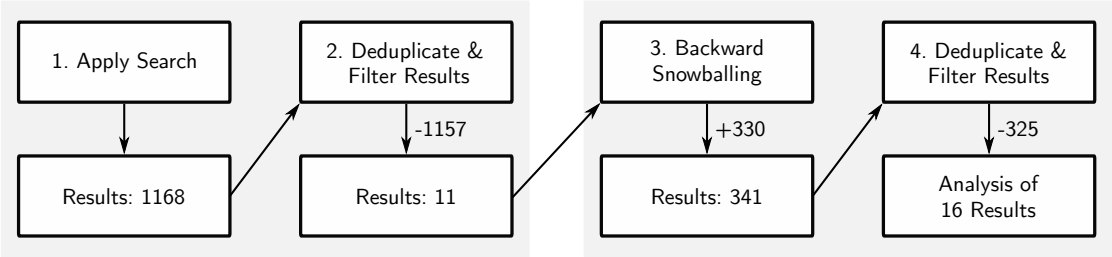


Figure 4.1: SMS process and amount of received publications.

4.1.1 Apply Search

Based on the research questions from RQ.1 we defined the following keywords: *test, code, model, readability, understandability, legibility* and *smell*. According to Oliveira et al. [50] the terms *readability, understandability and legibility* have overlapping meaning in software engineering, therefore we use them to create the concrete search strings shown in Table 4.1. Systematic mapping studies can be conducted with various search engines, which have a large search space like Google Scholar. We however use *Scopus*, *IEEE* and *ACM*, because they solely return peer reviewed papers [46], which increases trust in the quality of the SMS. Scopus is a meta search engine which indexes scientific literature of all kinds, ACM and IEEE are well-established sources for computer science literature.

The queries were performed on *Scopus*, *IEEE* and *ACM*, and we filtered the studies based on title, abstract and keywords. In the ACM string we removed the term "understandability" in the "abstract filter", because it returned too many results. For ACM we searched in the *ACM Guide to Computing Literature* which offers a larger search space than the *ACM Full-Text Collection*. We conducted the search at the end of November 2021 without limiting the publication year. It returned a total of 1168 raw results (Scopus: 458, IEEE: 230, ACM: 480), spanning from the year 2021 to 1969, with strongly decreasing studies per year before the year 2000. Based on the merged results, we proceeded to the next step.

4.1.2 Deduplicate & Filter Result

We first deduplicated the raw results based on the digital object identifier (DOI) and title, which removed 237 studies. Next, we imported the result set into a spreadsheet solution for applying inclusion and exclusion criteria. We **included** a study **if both** of the following criteria were fulfiled:

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

We **excluded** a study **if one** of following criteria applied:

Table 4.1: 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 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

We evaluated the criteria based on title and abstract of the results. When in doubt about including or excluding, the evaluated study was discussed with a second evaluator. This step left us with 11 results.

4.1.3 Backward Snowballing

Since relevant literature might refer to further important studies, we used the references included in the 11 studies for backward snowballing via Scopus. This increased the result set by 330 to a total of 341 studies.

4.1.4 Deduplicate & Filter Result

By comparing these 341 studies with the initial result set we found and removed 53 duplicates. Similar to step 2, we applied the inclusion and exclusion criteria. Additionally, after a full text reading, we summarised all remaining studies on presentation slides (see Figure 4.2 for an example) and discussed and reevaluated them as a author team. With this, we reduced the result set by 325 and obtained a final result of 16 studies.

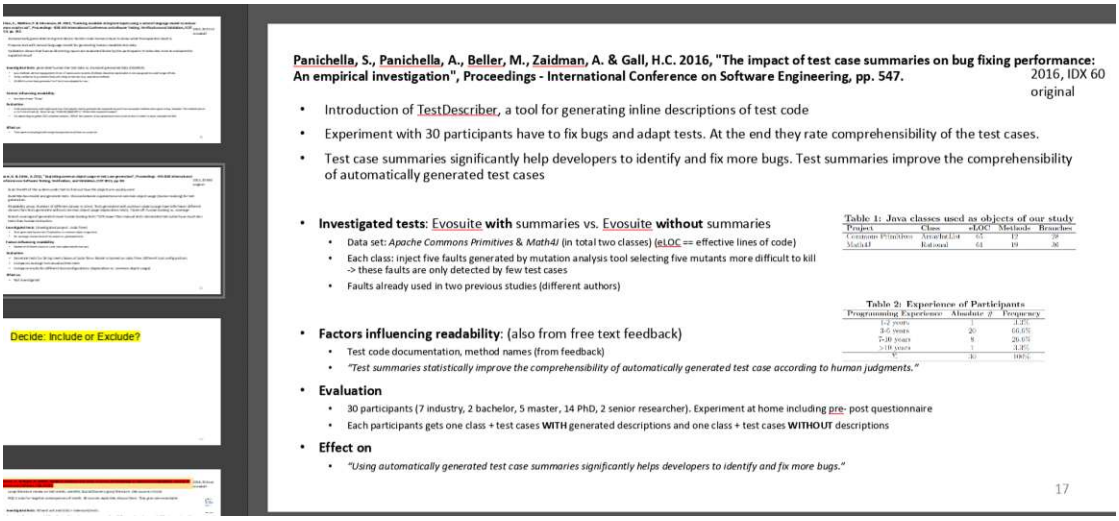


Figure 4.2: Example for a slide summarising contents of a study.

4.1.5 Excluded studies

In this subsection, we provide four representative examples and rationale for studies that we excluded in the final publication set after discussion of all authors:

Grano et al.[32] 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 was not in the scope of this work.

Tran et al. [69] 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. [3] 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[17] reports on a case study with emphasis on reporting experiences and challenges from a semi-automatic refactoring of a TTCN-2 test suite to TTCN-3. Improvements of readability focus on reducing complicated or unnecessary code artifacts generated by the automatic refactoring. We excluded this study since the focus was the migration from TTCN-2 to TTCN-3 and not an investigation of readability of test code.

Table 4.2: Final Set of Publications based on the Search Process.

Idx	Title	Authors	Venue	Year	Study Type	Analyzed Tests	RP
[A1]	Developer's Perspectives on Unit Test Cases Understandability	Setiani N. et al.[66]	ICSESS	2021	Experiment + Survey (hum)	Randoop, Evosuite, manual	
[A2]	DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests	Roy D. et al.[60]	ASE	2020	Experiment + Survey (hum)	Evosuite	yes
[A3]	Test case understandability model	Setiani N. et al.[65]	IEEE Access	2020	Experiment (hum)	Intellitest	
[A4]	On the quality of identifiers in test code	Lin B. et al.[44]	SCAM	2019	Survey (hum)	Evosuite, manual	yes
[A5]	An empirical investigation on the readability of manual and generated test cases	Grano G. et al.[30]	ICPC	2018	Experiment	Evosuite, manual	yes
[A6]	Specification-Based testing in software engineering courses	Fisher G. & Johnson C.[22]	SIGCSE	2018	Experiment + Survey (hum)	Spest, JMLUnit, manual	
[A7]	An industrial evaluation of unit test generation: Finding real faults in a financial application	Almasi M. et al.[2]	ICSE-SEIP	2017	Experiment + Survey (hum)	Randoop, Evosuite, manual	yes
[A8]	Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?	Daka E. et al.[13]	ISSTA	2017	Experiment + Survey (hum)	Evosuite, manual	
[A9]	How Good Are My Tests?	Bowes D. et al.[7]	WETSoM	2017	Concept paper (hum)	n.a.	
[A10]	Automatic test case generation: What if test code quality matters?	Palomba F. et al.[53]	ISSTA	2016	Experiment	Evosuite	
[A11]	Automatically Documenting Unit Test Cases	Li B. et al.[42]	ICST	2016	User study (hum)	manual	
[A12]	The impact of test case summaries on bug fixing performance: An empirical investigation	Panichella S. et al.[54]	ICSE	2016	Experiment (hum)	Evosuite	yes
[A13]	Towards automatically generating descriptive names for unit tests	Zhang B. et al.[76]	ASE	2016	Prototype and User Study (hum)	manual	
[A14]	Modeling readability to improve unit tests	Daka E. et al.[12]	ESEC/ FSE	2015	Experiment + Survey (hum)	Evosuite	
[A15]	Evolving readable string test inputs using a natural language model to reduce human oracle cost	Afshan S. et al.[1]	ICST	2013	Experiment (hum)	Test data: generated, manual	
[A16]	Exploiting common object usage in test case generation	Fraser G. & Zeller A.[24]	ICST	2011	Experiment	Generated (own solution)	

4.2 SMS Analysis Results

Table 4.2 summarizes the final set of studies obtained from the search process. The table provides index (*Idx*), *title*, *author*, *venue* and publication *year* of the studies. With the notion of research methods described by Wohlin et al. [74] in mind, it also provides information on the method reported by the study used for assessing readability (cf. column *Study Type* in Table 4.2) and if human participants were involved. The column *Analyzed Tests* indicates how the investigated tests were created e.g. manual (i.e., human written tests) or automated with a tool (in the table we mention the tool used for test generation). Finally, *RP* shows whether or not the authors offer their study material for replication.

4.2.1 Summaries of Selected Studies

In the following we briefly summarize the main aspects of each of the selected primary studies:

[A1] Setiani et al. [66]. In an online survey, 49 participants from freelancing platforms choose between two tests from a set of Randoop, EvoSuite or manual tests and comment on readability. Analysis of free text answers shows that naming (identifiers & tests), simplicity, independence (test is a unit test), structure, assertions (amount & message), comments and exceptions influence readability.

[A2] Roy et al. [60]. Deep learning model for generating test case summaries and renaming variables and test names. Approach applied to EvoSuite. In an online survey, 36 participants from academia and industry (some with knowledge of the system under test (SUT)) rate test cases. Results show that the improvement significantly increases readability.

[A3] Setiani et al. [65]. Test case understandability model by combining static code features identified by Daka et al. [A14] and developer related metrics (mostly experience) collected by an experiment conducted by Honfi and Mizskei [34]. In the referenced experiment, master students were asked to decide if a test case passes or fails. Results show that the combined metric performs better than the single metrics.

[A4] Lin et al. [44]. In an online survey, 19 subjects from academia and industry rate the quality of identifiers from test cases with possibility to suggest new identifiers. Results show several characteristics of various quality identifiers and a list of suggestions for identifier naming.

[A5] Grano et al. [30]. Comparison of production code, human tests, and EvoSuite tests with the readability model by Scalabrino et al. [62]. Investigated code from three Apache Commons projects with a total of 479 classes under test. Results show that in general human test code is less readable than production code, but generated test code is even less readable. Generated tests perform badly with respect to the metrics used by the readability model.

[A6] Fisher and Johnson [22]. Readability investigation of Java test cases generated with the tool Spest from formal specifications. In a survey, 134 students choose which of two tests is more readable. Results show that Spest tests are not as readable as human tests. The main difference between Spest and human tests lies in identifier naming and additional comments in the test.

[A7] Almasi et al. [2]. Evaluation of EvoSuite and Randoop by trying to detect already fixed faults in a real system and by investigating developers' opinions about these tools. Insights on the feedback include concerns about the readability of the generated tests, that the generated assertions only find easy faults, and that generated test data is not meaningful.

[A8] Daka et al. [13]. Generate test names based on the content of the test. Approach applied to EvoSuite. Comparison with human expert name in an online experiment with 47 students shows that the improved naming has a positive effect on students' ability to match names to test cases and production code. They also show that the participants agree with automatically generated names.

[A9] Bowes et al. [7]. During a two day workshop with industry partners a list of factors relevant to high quality tests was compiled. Enriched with guidelines on how to improve tests and how these factors can be assessed. Readability is one of these factors and is positively influenced by simplicity and expressiveness of tests. On the other side, the use of magic numbers and branching interfere with readability.

[A10] Palomba et al. [53]. Tool to compute test *cohesion* (i.e., "a test should only test one thing") and test *coupling* (i.e., "one behavior should only be verified by one test"), as defined by Meszaros [47]. Most EvoSuite tests perform badly on these metrics; therefore, the tool is added to EvoSuite's test generation as a quality measure to create improved tests.

[A11] Li et al. [A11]. UnitTestScribe is a tool for generating test code documentation or test case summaries for C# unit tests written by humans. An online survey with 26 persons from academia and industry shows that the generated descriptions are useful for understanding the test cases.

[A12] Panichella et al. [54]. TestDescriber is a tool for generating inline descriptions for tests. Experimental application on EvoSuite tests with 30 participants from academia and industry, which have to fix bugs, adapt tests, and rate comprehensibility of tests in the end. Results show that generated summaries support participants to identify and fix more bugs.

[A13] Zhang et al. [76]. NameAssist is a tool for renaming human written tests based on their contents. A comparison of BLEU scores from test names written by three graduate students, names from other name generators and NameAssist shows that NameAssist names are more similar to expert written names. A lab survey with three graduate students with 60 test cases shows that in 83% NameAssist names are at least equivalent

to the original name (of unknown quality). Names generated by NameAssist are mostly preferred over other name generators.

[A14] Daka et al. [12].By correlating various static code features with human readability ratings from crowdworkers for human written and EvoSuite tests, the authors create a readability model for test code. Features are amount of assertions, max line length, etc. The calculated correlations show several code metrics connected to readability. An improvement of EvoSuite with this readability model shows that 30 students decide 14% faster if a test passes, with no change in accuracy. Crowdworkers in an online survey prefer the improved tests in 69% of the cases.

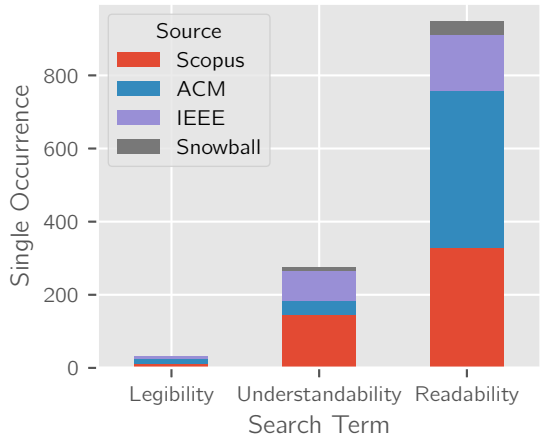
[A15] Afshan et al. [1]. Test data of type string automatically generated with respect to a natural language model. An online questionnaire with eight questions and participants from a crowdworking platform shows that it takes the participants less time to evaluate the expected result from a string method when human like test data is used. Comparison of their approach with IGUANA test data generator for C ported to Java.

[A16] Fraser and Zeller [24]. Tool for generating human looking tests by scanning the API of the SUT for common usage of objects in the system. The amount of imported classes in a test file is used as a readability proxy. Underlying rationale: when a test case needs objects from many different classes, it is harder to understand.

4.2.2 Contribution of Search Terms

In order to analyse the contribution of the search terms to the search result, we counted the amount of studies where the terms *readab, understandab* or *legib* occur in title, abstract or keywords. These word stems of the search terms are part of the actual search strings and account for different variations of these terms. Figure 4.3a shows the result for the raw unfiltered search results, Figure 4.3c gives the concrete numbers. Note that multiple search terms can appear in one study. Clearly *legibility* has the lowest contribution with 31 studies (2%) containing this term. This indicates that *legibility* is not used as a synonym for the other search terms in our result. *Understandability* appears in 275 studies (22%), however; we explicitly excluded this term in one part of the ACM search string, because it returned too many results. Still, Scopus and IEEE both have fewer studies with *understandability* than *readability*. Finally, *readability* has the largest contribution to the result with 949 studies (76%) containing this term.

The same kind of analysis for the final set of studies in Figure 4.3b shows a similar distribution. From the 16 studies none contain the term *legibility*, three (19%) *understandability* and ten (62%) *readability*. From this result we know that at least three studies do not contain one of the terms. This shows that snowballing yielded relevant studies, which could not be found by the original search string.



(a) Amount of studies from the **raw** unfiltered results containing search terms associated with readability.

(b) Amount of studies from the **final** set containing search terms associated with readability.

	Scopus	ACM	IEEE	Snowball	**Sum:**	**Perc:**
Legibility	12	12	7	0	31	2%
Understandability	147	36	83	9	275	22%
Readability	327	432	154	36	949	76%
Sum:	486	480	244	45	1255	100%
Perc:	39%	38%	19%	4	100%	

(c) Data for Figure 4.3a. One study can contain multiple search terms.

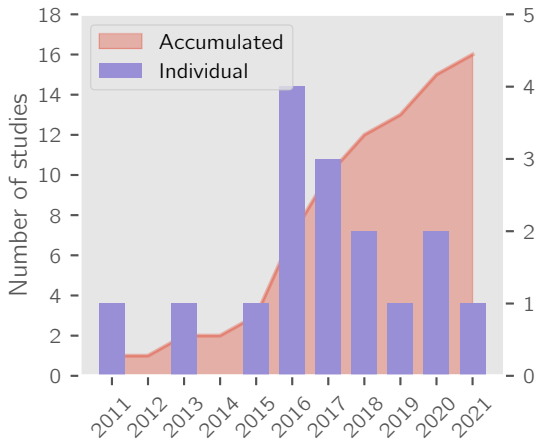
Figure 4.3: Contribution of search terms.

4.2.3 RQ 1.1. What is the importance of test code readability in scientific communities?

Venues addressing readability of test code. Table 4.2 shows 12 unique venues where studies were published and presented. Most prominently, three studies were presented at ICST (International Conference on Software Testing, Verification and Validation), two at ASE (International Conference on Automated Software Engineering) and at ISSTA (International Symposium on Software Testing and Analysis). The other venues each have one relevant study.

Timeline of published readability studies From the publishing years shown in Table 4.2 and visualised in Figure 4.4, we see that in 2016 four, 2017 three, and 2018 & 2020 two studies were published. In the other years spanning from 2011 until 2021 (exception to 2012 and 2014) one relevant study was published.

RQ 1.1 Findings. What is the importance of test code readability in scientific communities? The results indicate an ongoing general interest in the readability of test code (wide range of venues) with a strong focus on testing related venues and software engineering automation.



Years	Individual	Accumulated
2011	1	1
2012	0	1
2013	1	2
2014	0	2
2015	1	3
2016	4	7
2017	3	10
2018	2	12
2019	1	13
2020	2	15
2021	1	16

Figure 4.4: Number of studies per year and accumulated.

4.2.4 RQ 1.2 Which types of studies are published and which research methods are used?

Which types of studies are published? Table 4.2 gives an overview of the type of studies as reported by the authors of the respective papers. *Experiment* is the most prevalent type with a total of 12 studies containing either solely of an experiment (6) or an experiment in combination with a *survey* (6). A typical representative of the latter kind is Setiani et al. [A1], who conduct an A/B testing experiment with crowdworkers via a survey. While studies of type *experiment + survey* always contain human participants, studies with only an experiment do not involve humans in half of the cases (3). For instance, Grano et al. [A5] compared generated tests, (existing) human tests, and production code with ratings from a readability model. The remaining four studies each report a different kind of study. Lin et al. [A4] perform a *survey* on identifier quality in test code and Bowes et al. [A9] discuss testing principles in a *concept paper* from a workshop with industry partners. Li et al.[A11] conduct a *user study* for evaluating a tool for automatic test case documentation and, finally, Zhang et al. [A13] developed a *prototype* for test name generation, evaluated in a user study. Except from three studies, all involve some form of human participation where in nine cases also practitioners were included.

Which research methods are used in scientific studies? Concerning the utilized types shown in Table 4.2, most studies (12) report an experiment which is combined with a survey in 6 studies. Human involvement is quite common, in 13 from 16 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: When studies evaluate the effect of an approach with humans, they either ask participants to answer questions to a given test case without knowing the origin like in Roy et al. [A2] or Daka et al. [A8] or participants have to choose between two versions (forced choice) like in Setiani et al. [A1] or Daka et al. [A14]. For analysing the experiments results, seven from twelve 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. Two of these studies also use the Shapiro-Wilk test for normal distribution to decide if a parameterized test can be applied. The remaining studies interpret the results without statistical tests.

Survey: Five out of seven 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 seven 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. [A8].

User study and Prototype: The two studies of these types use surveys with Likert scale or forced choice questions with opportunity to elaborate on the rating. Zhang et al. [A13] 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: Bowes et al. [A9] 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.

RQ 1.2 Findings. Which types of studies are published and which research methods are used? The prevalent types of study are experiments which can contain a survey. Human participation in the experiments is common. 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.

4.2.5 RQ 1.3 Which factors influence readability of test code?

In this RQ we explore the factors that have been found to influence readability of test code. Table 4.3 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 (13 in total; see Table 4.3a) consider individual factors. Readability models were subject to study only in three instances (Table 4.3b), although such models are commonly used in the general research on source code readability.

In the following we briefly explain the factors we identified in the primary studies. The number in brackets shows the number of studies including the particular factor.

- **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, Zhang et al. [A13] propose a tool for automatic test renaming. Additionally, in some studies, e.g., like Setiani et al. [A1] or Bowes et al. [A9], participants agree on the importance of test names for readability.

- **Identifier names (5)**: Naming of variable names in test cases. Especially Lin et al. [A4] investigate this factor thoroughly and also provide characteristics of good and bad identifier names based on a survey. Roy et al. [A2] propose an automatic way for identifier renaming in test cases.

- **Comments (2)**: Single comments in test code providing useful information. According to Fisher and Johnson [A6], one of the differences between their generated tests and human tests is the lack of explanatory comments. In Setiani et al. [A1], survey participants also mention comments being to some degree important to readability.

- **Test summaries (3)**: Documentation describing the whole test case support understanding what the tests do, for example as Javadoc like in Roy et al. [A2] or interleaved with test code like in Panichella et al. [A12].

- **Test structure (5)**: Structural features of test methods like maximum line length, number of identifiers, length of identifiers, amount of control structures, etc. They are also used in combination by automatic readability raters e.g. from Daka et al. [A14], who propose a rater especially for test cases.

- **Dependencies (3)**: The number of classes a single test case depends on, as proposed by Fraser et al. [A16], or if a test is truly a unit test and therefore independent from other parts of the system. Test coupling and cohesion discussed by Palomba et al. [A10] describe dependencies between tests and are included in this factor.

- **Test data (3)**: 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. [A15] investigate this topic and show that readable string test data helps humans predicting correct outcome. Furthermore, in the workshop study from Bowes et al. [A9] developers, amongst others, state that *magic numbers* are detrimental to readability.

- **Assertions (2)**: 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. [A14] report low correlation and predictive power for the amount of assertions in a test with respect to readability. In the survey from Setiani et al. [A1] assertions are mentioned to have an influence, but other factors like naming are deemed more important. Almasi et al. [A7] report developer concerns on automatically generated assertions.

• **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. [62] used by Grano et al. [A5].

Table 4.3: Reported factors influencing test code readability.

(a) Studies investigating individual factors.

Individual factors	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16
Test names (6)	•	•						•	•			•	•			
Identifier names (5)	•	•		•		•			•							
Comments (2)	•					•										
Test summaries (3)		•									•	•				
Test structure (2)	•								•							
Dependencies (3)	•									•						•
Test data (3)							•		•						•	
Assertions (2)	•						•									

(b) Studies using readability models.

Readab. models	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16
Textual features (1)					•											
Test structure (3)			•		•									•		

RQ 1.3 Findings. Which factors influence readability of test code? The majority of studies investigates influence of individual factors on readability. Overall the top three of investigated factors are test names, test structure and identifier names

4.2.6 RQ 1.4 Which type of test code was investigated in context of readability?

This RQ explores the type of test code investigated in the primary studies, e.g., in terms of test level, programming language, and generated/automated code. Except

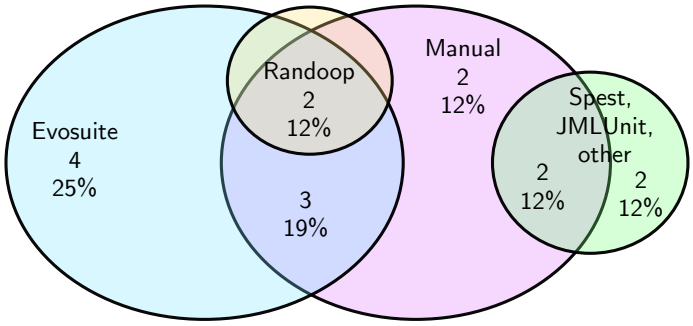


Figure 4.5: Venn diagram showing combinations and amount of different types of tests analyzed by the relevant studies.

from Li et al.[A11] and Setiani et al. [A3], who investigate C# unit tests, the dominant programming language is *Java*. Unit tests are the primary target with respect to the testing level although improvements for, e.g., identifier naming could also be applied to tests at other levels.

Figure 4.5 shows the studied types of tests and the combinations in which they were compared. 9 studies in total (4+2+3) investigate EvoSuite, either individually or in combination with other generated or manually written tests. Randoop tests are investigated in 2 studies in combination with EvoSuite and manual tests. 4 studies investigate tests generated with other tools. Although manual tests are exclusively investigated only 2 times, they appear 9 times in total including combinations with generated tests.

> *RQ 1.4 Findings. Which type of test code was investigated in context of readability?* The main focus of readability investigation lies on automatically generated tests, which are often compared to manually written tests.

4.2.7 RQ 1.5 Executable Readability Models

The following sections describe the search for a readability rating tool based on the raw search results of the previous academic SMS. Figure 4.6 gives an overview on the individual steps.

Step 1: Apply Search. We used the raw search results obtained from search queries on Scopus, IEEE and ACM as described in Section 4.1.

Step 2: Deduplicate & Filter Results. We filtered the search results of the academic SMS with focus on tools for rating readability of source code. We **include** a study **if both** of the following criteria were fulfilled:

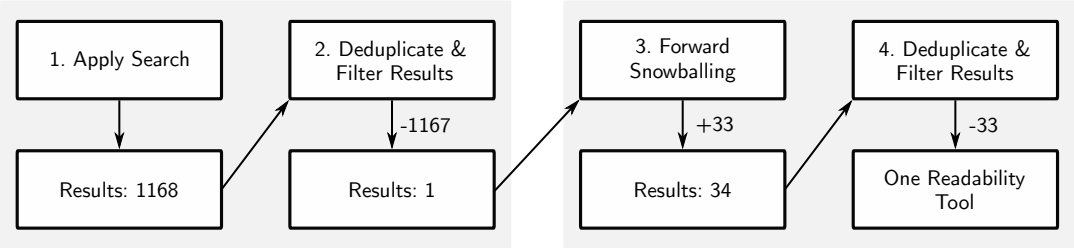


Figure 4.6: Tool search process.

- Conference papers, journal/magazine articles, or PhD theses (returned by ACM)
- The study provides access to a tool for rating readability of source code.

We **excluded** a study **if one** of following criteria applied:

- Not written in English
- Conference summaries, talks, books, master thesis
- Duplicate or superseded studies

By analysing title and abstract the authors reduced 1168 studies to 16 for full text reading. The 16 studies were selected, because they may use or propose a model for rating readability of source code. After full text reading, one study by Scalabrino et al. [64] remains, because it is the only study which provides access to the proposed readability rating tool.

Step 3: Forward Snowballing. Since the search string for the academic SMS targets studies on test code readability, there may exist other tools which generate readability ratings of source code. In order to find such tools, we use the single remaining study for forward snowballing with Scopus, which returns 33 studies citing Scalabrino et al. [64] in September 2022. We think that this approach is reasonable, because 5 out of the 16 studies already use the tool by Scalabrino et al. presented in [64] (or a preceding paper e.g. [62]) in their work. This indicates that this tool is known in the community. Therefore, we assume that a study proposing a new tool for rating readability will reference Scalabrino et al. [64].

Step 4: Deduplicate & Filter Results. None of the 33 studies present executable tools for rating readability of code snippets. There exists a study which may provide an executable tool in the future. Karanikiotis et al. [37] present examples for readability

ratings on a website [1], which may allow to rate readability of arbitrary code snippets in the future. At the time of writing (September 2022) this functionality is not implemented.

Result Since the forward snowballing did not add new tools to the result set, we will use the tool by Scalabrino et al. [64] for the rest of this work.

> *RQ 1.5 Findings. Which executable models for assessing code readability exist?* We found one tool by Scalabrino et al. [64], which can be used for assessing code readability.

[1] Readability rating website by Karanikiotis et al. https://readability-evaluator.netlify.app/

CHAPTER 5

Grey Literature Study

In this chapter, we first describe the study protocol and process, with the used search string and search engine for the grey literature study followed by presentation of the results including a list of readability factors discussed in practice and a comparison to the results of the previous systematic mapping study of scientific sources in Chapter 4.

Note: This chapter is part of a work, which is currently under review for publication as Winkler et al. [73].

5.1 Study Protocol and Process

The process for conducting the review of grey literature is similar to the scientific literature review, except that there is no backward snowballing as shown in Figure 5.1. A key guideline and source of information was the work by Garousi et al. [29], which provided many useful inputs for this part of the work. We decided to add grey literature to this work, because testing is primarily done in industry by practitioners. Instead of asking testers in an survey about their preferences, we collected various sources on the internet, because we assume that the internet is one of the first places used for information gathering.

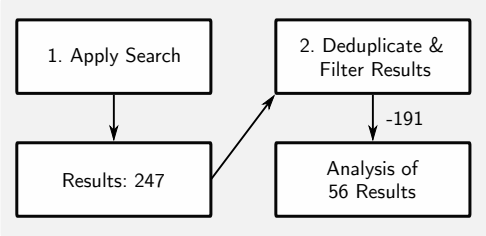


Figure 5.1: Grey literature review process and amount of received grey sources.

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, the search returned 146 results for *"test code" readability* and 101 for *"test code" understandability* (total: 247) in mid-February 2022.

Step 2: Deduplicate & Filter Results. We first deduplicate the results by comparing the links which removed 11 sources. The result set was imported into a spreadsheet solution for applying inclusion and exclusion criteria.

Inclusion Criteria. We *included* a source *if both* of the following criteria were 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 source *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 is on the one hand the information returned by the search, because we collected all information Google displays on its result pages. On the other hand we visited each page where the given information was not sufficient for a clear decision. This step left us with 56 results ready for further analysis.

Excluded Sources. As for the scientific literature, we provide some examples and rationale for sources, that where excluded when applying in and exclusion criteria: The entry of Karhik [1] 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 reasons for this claim. In total this source reads like an advertisement for AssertJ. The entry of Bas Broek[2] is a large blog entry with a primary focus on the

[1] 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/

[2] Bas Broek, (Improving Your) XCTAssert* Failure Messages | Bas' Blog, https://www.basbroek.nl/xctassert-asterisk

readability of assertion failure messages. Factors relevant to this study naming of test cases and a given-when-then structure are mentioned in a total of three sentences which is only a small proportion of the whole entry. With the same rationale we also exclude source Wikipedia[3], because the primary focus is on test driven development and readability is a side topic. Likewise the entry on Programmer All[4] has much content and also provides code snippets. Still, the focus is on unit testing in general and not improving readability. Karlo Smid[5] discusses the DRY-principle (don't repeat yourself) in the context of unit testing. However, the blog entry is relatively short and primarily references to another source already present in the result set [G51]. Although the following collaborative source[6] has a reasonable size and also has a section on readability, the statements are too generic and do not contain a concrete influence factor to readability. Finally, there are also many sources which are off-topic, because e.g. they discuss general code readability or quality; advantages of unit testing; or are documentation pages of test frameworks.

5.2 Grey Literature Analysis Results

In the following sections we present the results[7] of the grey literature review and give answers to RQ2.1 and RQ2.2.

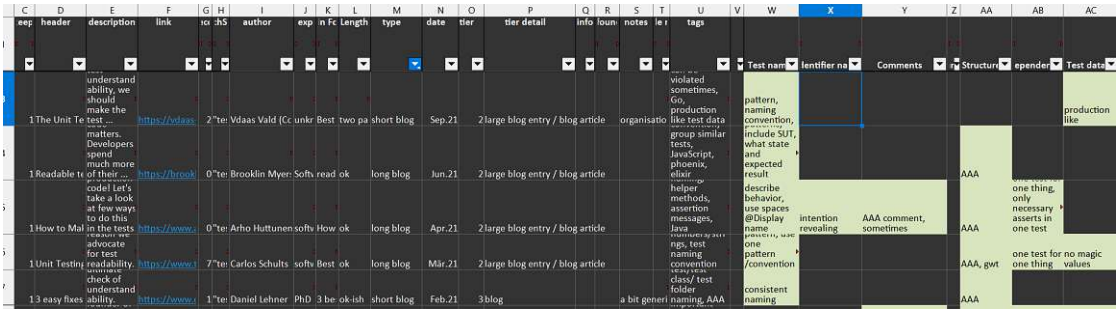


Figure 5.2: Snippet from grey literature analysis showing sources and extracted readability factors.

We extracted the discussed readability factors by tagging each source with keywords which are mentioned in the context of test code readability, see Figure 5.2 for a snippet of the spreadsheet. If a keyword is mentioned in a different context it is excluded. For example in [G41] the use of helper methods is only mentioned in context of easier maintenance, therefore this appearance of helper methods is not counted.

[3] Wikipedia, Test-driven development, https://en.wikipedia.org/wiki/Test-driven_development

[4] Unit test 2 - Programmer All, https://www.programmerall.com/article/98141652585/

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

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

[7] The availability of the internet sources can not be guaranteed. At the time of the grey literature mapping mid-February 2022 all the sources listed in this work where available.

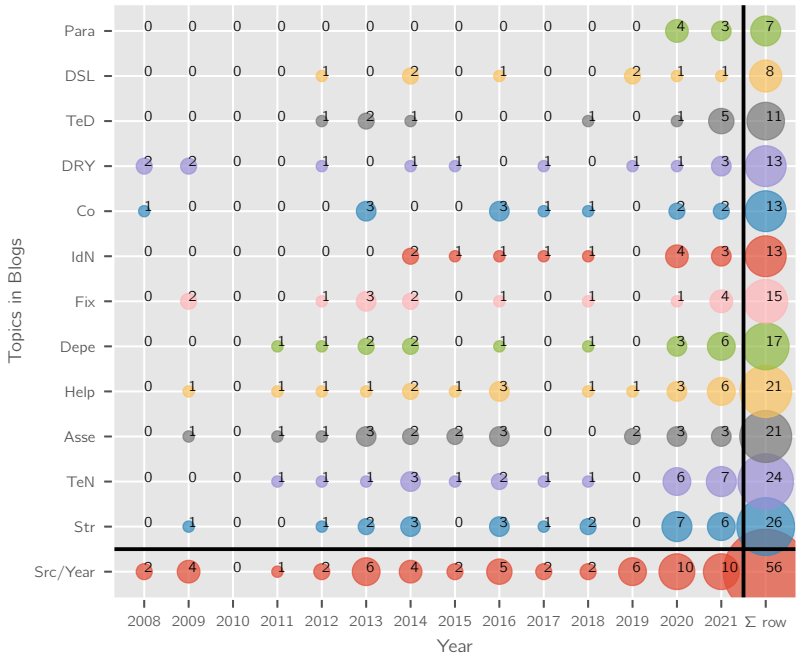
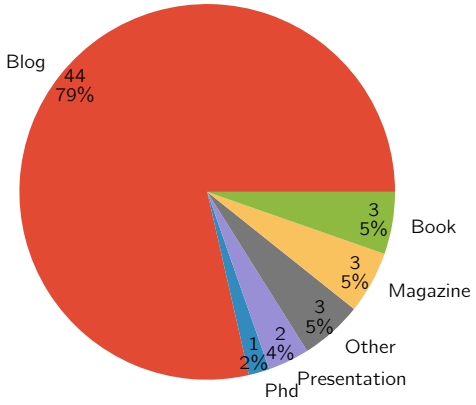


Figure 5.3: Factors investigated by grey literature. The bottom most row gives the amount of sources per year, which may investigate multiple factors.

Factor across years. Figure 5.3 shows the factors investigated by the blogs across the years. The bottom line *Src/Year* gives the number of sources in this year which investigated the factors above. Some sources discuss multiple readability factors, hence the sum of discussed factors is larger than the sum of sources. Apart from parameterized tests, which appeared seven times only in the years 2020 and 2021 and fewer sources in 2017 and 2018 there are no obvious fluctuations in the distribution of factors. Table 5.1 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 16 sources mention Java or use Java code snippets, C# appears in nine and Java Script in eight sources. Kotlin, Python and Ruby each appear in two sources, Scala, Typescript and Go are mentioned in one source each. Some sources do not mention a certain programming language or use code snippets, because they provide general best practices for testing. This is in accordance with the findings in academic literature in the previous chapter in Subsection 4.2.6, where Java is the dominant language used in studies on test code readability.

Source types. Figure 5.4 shows the identified types of grey source. From the 56 sources we identify around 80% (44 in total) as blog entries of various sizes. A source is also identified as a blog, when there is no clear indication that an editorial



Type	Absolute	Relative
Blog	44	79%
Book	3	5%
Magazine	3	5%
Other	3	5%
Presentation	2	4%
Phd	1	2%
Sum:	56	100%

Figure 5.4: Identified types of the selected grey sources.

team is involved. The types of the remaining 12 sources are spread out quite evenly across three books, three magazines, two presentations (slide shows), one Phd thesis and three other types (stackoverflow, wiki, cheatsheet).

5.2.1 RQ 2.1 Which influence factors are discussed in grey literature?

Table 5.1 shows all selected grey sources and their associated factors. The first set of factors originate from the previous systematic mapping study (SMS) on scientific literature shown in the previous chapter in Subsection 4.2.5. The second set of factors were only found in the grey literature and did not appear in the previous SMS. The detailed investigation on the differences follows in Subsection 5.2.2.

Known Factors from SMS: This set of influence factors have been derived from the SMS with focus on whether or not these factors have been discussed in practice.

Assertions (21) (Asse): The use of appropriate assertions or custom assertions is suggested in eleven sources e.g. [G7][G2]. 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 [G43][G12]. Four sources stress the importance of assertion messages for debugging.

Table 5.1: List of all selected sources by years descending found by the grey literature search, mapped to factors relevant to readability. Orange factors were already found in the previous SMS. **Asse**: assertions, **Co**: comments, **Depe**: dependencies, **IdN**: identifier names, **Str**: structure, **TeD**: test data, **TeN**: test names, **TS**: test summaries,, **TF**: textual features, **DRY**: DRY principle, **DSL**: domain specific language, **Fix**: fixtures, **Help**: helper structures, **Para**: parameterized test

	Asse	Co	Depe	IdN	Str	TeD	TeN	TS	TF	DRY	DSL	Fix	Help	Para
[G2]	•		•	•		•	•					•	•	
[G35]	•		•		•						•	•	•	
[G45]										•			•	
[G21]			•	•		•	•			•		•	•	•
[G55]		•	•		•	•						•	•	•
[G53]						•	•			•				•
[G31]					•		•							
[G23]	•	•	•	•	•		•						•	
[G36]					•		•							
[G48]			•		•	•	•							
[G3]			•	•	•	•	•							•
[G6]	•			•	•		•							
[G22]										•				•
[G39]			•	•	•		•						•	
[G20]	•	•			•								•	
[G54]				•			•							
[G19]					•		•							•
[G16]					•						•	•		•
[G49]		•	•		•		•							
[G50]	•												•	
[G51]										•				
[G43]	•													
[G1]	•												•	
[G30]											•			
[G34]											•			
[G47]														
[G46]			•		•									
[G44]		•		•	•	•	•					•	•	
[G17]		•								•				
[G38]				•	•		•							
[G52]							•							
[G12]	•	•			•						•		•	
[G41]				•			•							
[G18]	•	•	•		•							•	•	
[G13]	•	•			•								•	
[G42]	•													
[G9]	•			•			•			•			•	
[G15]			•	•	•		•				•	•		
[G56]	•				•		•					•	•	
[G5]					•					•				
[G26]	•		•	•		•	•				•		•	
[G4]	•	•												
[G29]	•		•			•						•		
[G8]	•		•			•						•		
[G33]		•			•		•							
[G32]														
[G11]		•			•							•	•	
[G27]			•		•					•		•		
[G14]	•					•	•				•		•	
[G7]	•		•				•						•	
[G40]												•		
[G10]										•		•		
[G24]	•													
[G37]					•					•			•	
[G25]										•				
[G28]		•								•				
Sum:	21	13	17	13	26	11	24	0		13	8	15	21	7

Comments (13) (Co)): Eleven sources use comments in their snippets or also 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 [G49] uses empty lines as an alternative or [G12] 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. [G18].

Dependencies (17) (Depe): All 17 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 highlight to only assert properties which are absolutely necessary for the functionality described by the test name and resist the urge to check additional properties. The 'one assertion per test' rule mentioned by e.g. [G23][G18] can be tried out but often a functionality has to be checked with multiple assertions.

Identifier names (13) (IdN): While seven sources only give generic information (e.g. *should have meaningful or intention revealing names*), other authors suggest to either prefix variables with *expected* and *actual* [G21] or use names like *testee, expected, actual* [G9]. Overall the naming should be consistent.

Structure (26) (Str): 21 out of 26 sources suggest the use of patterns like *Arrange, Act, Assert* ([G23]), *Given, When, Then* [G12] or *Build, Operate, Check* [G46]. Two sources [G27] [G18] suggest to group similar test cases to see difference more quickly. Sources [G44][G16] 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 different authors.

Test data (11) (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. [G29][G26]. However, [G21] 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. Finally, test data should be production-like and simple, one author also recommends to highlight important data.

Test names (24) (TeN): All sources suggest coherent naming of test cases and most of them suggest a concrete naming pattern like *givenFooWhenBarThenBaz* [G2] or *subject_scenario_outcome* [G49]. Still many acknowledge that there exist various patterns as alternatives. Three sources which cover Java and Kotlin [G23][G52][G21] explicitly suggest to use spaces in test names. This may be caused by the lack of support for such naming by Java, since other authors using other languages like Ruby or Javascript use spaces in their test names without further note e.g. [G49][G3]. Long names are explicitly okay for two sources, since these methods are

not called in other parts of the code. Finally, different opinions exist on the inclusion of the name of the concrete tested method in the test name. Sources [G33][G9][G7] 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. On the other hand side [G14] and [G26][G44] suggest to include the method name in the test name.

Test summaries (TS), **Textual features** (TF) **(0)**: Test summaries were not mentioned in any of the sources. For textual features, which focus on natural language aspects in test cases, we could argue that using a consistent naming scheme falls into this factor. However, we already have specialised factors which are more suitable.

New Factors from Grey Literature Analysis: This set of influence factors have been discussed in practice with limited consideration in scientific literature.

DRY principle (13) (DRY): In the sources which mention the **D**on't **R**epeat **Y**ourself principle there is an agreement that strict adherence to this principle hides away information important for understanding test cases. Instead some suggest to focus more on the **D**escriptive **A**nd **M**eaningful **P**hrases (DAMP) principle [G9] or to find a balance between these principles. Sources [G28][G45] e.g. suggest to describe what is done thoroughly (DAMP), how it is done can be hidden in a helper method (DRY).

Domain specific language (8) (DLS): In order to make tests more readable also for non programmers these authors use helper functions or Gherkin (behavior driven testing with Cucumber) as domain specific languages. These languages describe the executed behavior in a natural language way and hide away the execution details e.g. [G30][G34].

Fixtures (15) (Fix): Although 13 authors use fixtures, sometimes in combination with setup methods, two authors [G56][G44] argue against the use of fixtures, because they are not visible in the test itself and may contain important information. Similarly [G21] argues that moving reusable test data into a fixture forces the reader to jump between two locations. Finally, [G15] suggest that fixtures should only be used for infrastructure and not for the system under test.

Helper structures (21) (help): 13 sources recommend helper methods in order to hide (irrelevant) details like creating objects or asserting properties [G20][G13]. The *Builder Pattern* (or similar patterns) are used by five sources for creating the objects under test e.g. [G37][G55]. Inheritance of test classes is seen critically by three authors e.g. [G45][G56].

Parameterized test (7) (Para): The sources use parameterized (aka data-driven or table-driven) tests to reduce code duplication. This is also done by authors who are not in strict favor of the DRY principle e.g.[G21].

RQ 2.1 Findings. Which influence factors are discussed in grey literature? Most of the sources discuss structural influence factors, test naming, usage of assertions and helper methods and test dependencies.

5.2.2 RQ 2.2 What is the difference between influence factors in scientific literature and grey literature?

Most of the factors found in scientific literature are also mentioned in grey sources. However, the focus of the investigation can vary. E.g., grey literature focuses on semantic structure like *Arrange, Act, Assert*, whereas scientific literature has a stronger focus on enumerable structural properties like line length or number of identifiers. Table 5.2 gives an overview on the differences. In the following section we compare views from the scientific community and practitioners.

Assertions (21) (Asse): There is little evidence in scientific literature on the effect of assertions on readability. Setiani et al. [A1] report low influence of assertion messages. Almasi et al. [A7] report concerns from developers about the meaningfulness of generated assertions. Leotta et al. [41] report no significant influence on test comprehension when AssertJ is used instead of basic JUnit assertions. However, other positive effects were observed.

Comments (13) (Co)): The usage of comments for highlighting the structure of the test is not investigated in scientific literature. Fisher and Johnson [A6], explain different readability ratings between generated tests and human tests also with the lack of explanatory comments. Setiani et al. [A1], survey participants also mention comments being to some degree important to readability. These findings are to some extent surprising, because the recommendation in grey literature such explanatory comments is generally to avoid them.

Dependencies (17) (Depe): The recommendation that one test should only test one functionality or behavior is added to test case generation by Palomba et al. [A10]. According to Daka et al. [A14] and Setiani et al. [A1] the amount of assertions only has low predictive power for readability. This supports the not so strict interpretation of the "one assertion per test" rule.

Identifier names (13) (IdN): The survey from Lin et al. [A4] shows the importance of meaningful, concise and consistent identifiers. The renaming approach by Roy et al. [A2] also suggests variable names like *expected* and *result*. The deep learning model was trained with software projects reaching a certain level of quality. Therefore, it appears like identifier names as mentioned by the grey literature sources are a part of high quality tests.

Table 5.2: Differences in influence factors between scientific and grey literature. (Overlapping factors shown as highlighted rows.)

Formal Literature	Grey Literature
Structure	
Identifier length	
Line length	
Constructor calls	
Unique identifiers	
Number of identifiers	
Control structure	Control structure
Length of test case	Length of test case
Other static code features	
	Avoid eye jumps
	Group similar test cases
	Coherent formatting
	Semantic structure (AAA, GWT, etc.)
Test names	
Use of patterns	Use of patterns
	Consistent naming
	Long names okay
	Spaces in names
	Include method under test in name
	Do not include method under test in name
Assertions	
Amount of assertions	
	Fluent assertions
Assertion messages	Assertion messages
	Appropriate assertions
	Custom assertions
Helper methods /classes	
	Builder pattern
	Avoid inheritance of test classes, prefer composition
	Methods for each step (given when then)
	PageObject
Dependencies	
One test for one behaviour	One test for one behaviour
Amount of assertions	(Try) one assert per test
Setup method / fixtures	
	Use both
	Use setup methods, avoid fixture
	No test data in fixture
	Avoid too long fixtures
Comments	
	For structuring test cases (e.g. AAA pattern) (or use empty lines)
Explanatory comments	Avoid, can become outdated
DRYness	
	Not too DRY, violate if needed
	DRY - DAMP balance
	KISS
Identifier names	
Consistent	Consistent
Concise	Concise
Meaningful	Meaningful
Use patterns	Use patterns
Test data	
No magic values	No magic values
Production like / simple values / human like	Production like / simple values / human like
	Hard coded expected values instead of computed
	Highlight important data

Structure (26) (Str): Literature published in academic context focuses more on countable properties like maximum line length, amount of control structures etc. e.g. Grano et al. [A5], Daka et al. [A14] or Setiani et al. [A3]. In contrast to this, the authors of the grey sources focus on a semantic form of structure like the AAA pattern, which is also discussed in another study by Setiani et al. [A1]. They report moderate positive influence on readability from the *Arrange, Act, Assert* structure.

Test data (11) (TeD): Participants of the workshop from Bowes et al. [A9] also recommend to avoid magic values. Almasi et al. [A7] and Afshan et al. [A15] highlight importance of meaningful or human-like test data.

Test names (24) (TeN): Like in grey literature, scientific literature also uses naming patterns, when test cases are renamed. Zhang et al. [A13] or Daka et al. [A8] use *testSubjectOutcomeScenario* although *outcome* and *scenario* can be left out. The approach by Roy et al. [A2] generates test names with a machine learning model. Based on the examples given in the paper it does not seem to include the concrete method under test in the name. In other studies e.g. by Panichella et al. [A12] or Setiani et al. [A3] survey participants highlight the importance of meaningful test names.

Test summaries (TS), **Textual features** (TF) **(0)**: Test summaries were not mentioned in any of the sources. For textual features, which focus on natural language aspects in test cases, we could argue that using a consistent naming scheme falls into this factor. However, we already have specialized factors which are more suitable.

RQ 2.2 Findings. What is the difference between influence factors in scientific literature and grey literature? There is a clear intersection in the factors investigated by scientific and grey literature. However, the factors are sometimes covered in different ways. We found two influence factors exclusively covered in scientific literature and extracted five new influence factors from grey literature.

CHAPTER 6

Initial Readability Study

For the following initial readability study we take the results from the systematic mapping study (Chapter 4) and the grey literature review (Chapter 5) and investigate a selection of identified influence factors influencing with focus on the perception of test case readability. Materials of the experiments are available.[1]

Note: Parts of this chapter are part of a work, which is currently under review for publication as Winkler et al. [73]. The parts concerning the rating criteria of students (Subsection 6.2.3), the analysis of the impact of experience on readability ratings (Subsection 6.2.4) and the comparison to the readability rating tool (Subsection 6.2.5) are novel in this work.

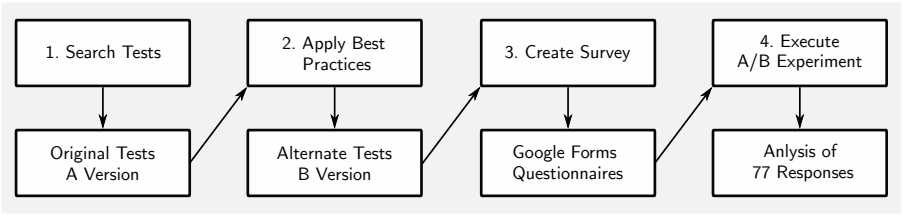


Figure 6.1: Experiment process and amount of received responses.

6.1 Experiment Setup and Procedure

Participants from multiple years of a master course on software testing at TU Wien were invited to participate in this online challenge, with the possibility of a few bonus points for the course as a reward.

[1] Material available under: https://drive.google.com/drive/folders/1pvFgQ4md2_Gpthr_fx6x0H8vTiu89Qly?usp=share_link

Table 6.1: Listing of test cases with their assigned influence factor, originating project and differences made for both versions. **A** (original version) and **B** (altered version) denote the groups.

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)
Assertions	showsAllStsGaDownloads	Dchartfield Sagan	Specific assertions (JUnit vs. Hamcrest/AssertJ)
Assertions	indexedReadAndIndexed WriteMethods	Spring Framework	Specific assertions (JUnit vs. Hamcrest/AssertJ)
Dependencies	testChoicesWithValid DefaultValue	Apache Flink	Remove unnecessary try catch
Dependencies	testApplyToMovesValue PassedOnShortName ToLongNameIfLong NameIsUndefined	Apache Flink	Remove unnecessary try catch
Dependencies	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

The experiment follows an A/B testing approach, because such experiments are in general a good approach for comparing the effect of a treatment to a population. Therefore, the participants rate readability of original and altered test cases. In our scientific literature review we also found some studies using this approach e.g. Roy et al. [60] or Setiani et al. [66]. Figure 6.1 shows an overview on the experiment process. We discuss the individual steps in the following sections.

Search Tests

We selected 27 test cases covering different influence factors from 8 sources, which also contain generated tests by Randoop and Evosuite. Table 6.1 shows influence factor, test name and origin project. Additionally we selected 3 tests as control group, which are not shown in this table. Most tests including the automatically generated tests come from *Apache Commons Lang3*. The last three tests with origin project "Student Solution" are selected tests written by students for an course assignment.[2]

Apply Best Practices

For each test, except the control groups, we create alternative versions of these tests, keeping in mind the results from the previous findings and our own experience. Column *"Modification A/B"* in Table 6.1 gives a short description on the differences between A and B version.

Create Survey

For each challenge run we use a selection of 18 original test cases, highlighted by the horizontal line in Table 6.1 to create the questionnaires for the survey. In total there are 6 different questionnaires consisting of 3 where the test cases are in "original" order and 3 where this order is reversed. Each questionnaire contains tests from each influence factor and for both versions, the participants do not know of influence factors or if the test was modified. Also they either get to see the original test case or the modified version i.e., they never see both versions of the same test case. Including tests of the control group each questionnaire contains 12 tests in total. The participants are asked to rate the readability on a 5 point Likert scale from 1 (unreadable) to 5 (easy to read) and to give up to three free text reasons for their rating. Before and after this main task, there is a pre- and post-questionnaire for collection information on the participants and acceptance of the challenge.

We created the questionnaires *google.forms*, because it provides an easy way for creating surveys, which can also be reused for future replications. Although the graphical user interface for creating the forms is intuitive and easy to use, importing the screenshots of the test cases or reordering the questions to create the reversed versions is tedious and error-prone. Luckily Google offers a JavaScript based scripting language (*Apps Script*) for

[2]Material available under: https://drive.google.com/drive/folders/1pvFgQ4md2_Gpthr_fx6x0H8vTiu89Qly?usp=share_link

a range of its services, which we use to automate this part of questionnaire creation. The collected data from the forms can be exported in various formats for further processing. Beside the survey forms, we provided the selected tests in a PDF and as plain text files as additional materials for the study participants.

Execute A/B Experiment

The survey was open for two weeks in each iteration and the participants were free to start and stop their run at any time in this period. The working duration for one round was about one hour. Across all iterations, we received responses from 77 participants.

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 detected, we report the effect size with Cliff's Delta (δ). Roy et al. [60] used the same approach for their Likert scale data. Cliff's Delta is interpreted according to Romano et al. [59] with $|\delta| < 0.147$ *"negligible"*, $|\delta| < 0.33$ *"small"*, $|\delta| < 0.474$ *"medium"*, otherwise *"large"*

For the qualitative analysis of the free text comments we use the previously found influence factors as base factors. In order to keep more detailed information of the comments during categorisation, we add sub-factors. These sub-factors are added each time, when a comment does not fit into the already existing factors. Comments are assigned to every fitting category, hence one comment can appear in multiple categories. The categorisation was done in a SVG-Editor in a hierarchical structure (see Figure 6.2) and analysed with a spreadsheet solution.

6.2 Experiment Results

This section presents the results of the controlled experiment to investigate the readability of a selected set of test cases. Some factors influencing readability appear more than once in Table 6.1 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.

6.2.1 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 6.2 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

Figure 6.2: Example screenshot from comment categorisation. The comments are stacked beside the sub-factors. Yellow comments are duplicates.

Table 6.2: Information on participants experience.

(a) General Software Development Experience [years].

Years	Absolute	Percentage
2-5	41	53.2%
>5	34	44.2%
1-2	2	2.6%
Sum:	77	100.0%

(b) Professional Software Development Experience [years].

Years	Absolute	Percentage
2-5	25	32.5%
1-2	24	31.2%
0	20	26.0%
>5	8	10.4%
Sum:	77	100.1%

around 30% have either one to two or two to five years of experience. In total around 75% have worked at least one year.

6.2.2 RQ 3.1 Do factors discussed in practice show an influence on readability when scientific methods are used?

Figure 6.3 shows the distribution of the aggregated readability ratings including boxplots for the investigated modification mapping to influence factors. Table 6.3 shows the results from the statistical analysis. The first column "Modification A/B (Influence Factor) maps to the according columns in Table 6.1. We discuss each modification in the following sections. As a reminder, we interpret Cliff's Delta (δ) according to Romano et al. [59] with $|\delta| < 0.147$ *"negligible"*, $|\delta| < 0.33$ *"small"*, $|\delta| < 0.474$ *"medium"*, otherwise *"large"*,

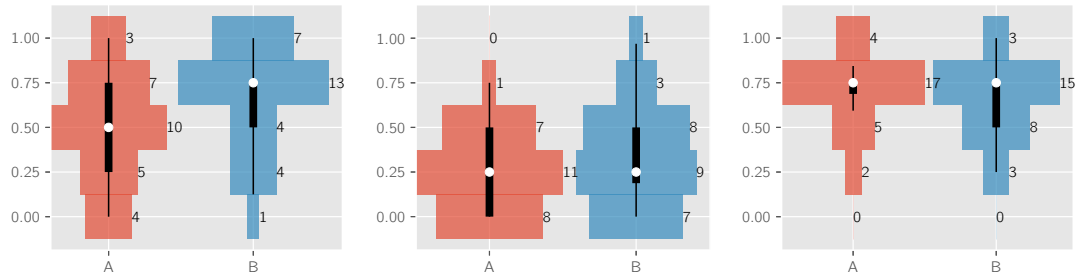
Loops vs. Unrolled (Figure 6.3a). In this modification the difference between A and B of the aggregated results is significant with $p = 0.02$. The effect size $\delta = -0.35$ is on the lower end of a *"medium"* effect size. The analysis of the individual tests reveals that the whole modification is 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 (Figure 6.3b). Here the difference between A and B is barely significant $p = 0.04$ for the second test, 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 (Figure 6.3c). Neither the figure nor the statistical analysis show a significant difference in the ratings.

Structure (Figure 6.3d). 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 $p = 0.16$.

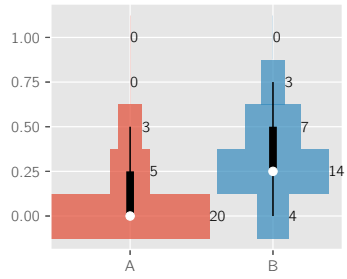
Comments (Figure 6.3e). Although none of the individual tests has a significant difference between A and B, the aggregated result is significantly different with $p = 0.02$ and has a lower *"medium"* effect size with $\delta = 0.36$. Since we removed comments in the original versions of the tests, the A version contains more information than B. A look at Figure 6.3e and the median values in the Table 6.3 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.



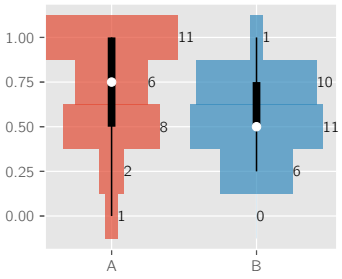
(a) Loops vs. Unrolled

(b) Try Catch vs. Asserts

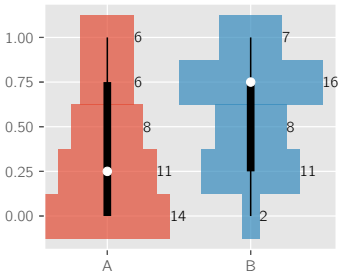
(c) Variable Reuse



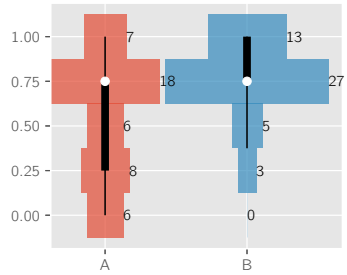
(d) Package Names, If-Str.



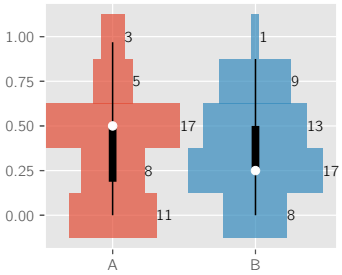
(e) Remove Comments



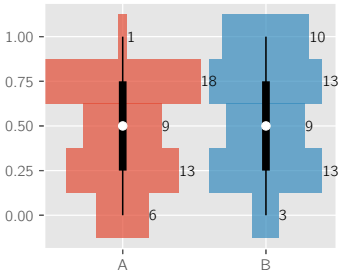
(f) Loops vs. Parameterized



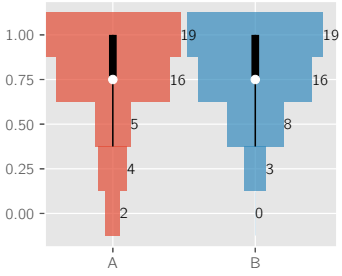
(g) Split Up Tests



(h) Specific Assertion



(i) Unnecessary Try Catch



(j) Remove Fixture

Figure 6.3: Distribution and box plots of aggregated readability ratings per A/B modification. Ratings from a five-point Likert scale range from 0 (not readable) to 1 (very readable). The numbers on the right hand side of the histograms represent the amount of answers for this rating.

Table 6.3: Statistical analysis of experiment results using a two-sided Wilcoxon Rank Sum test (p) and Cliff's D (δ) for effect size. δ is only shown for $p < 0.05$.

	A			B			Compare	
Modification A/B (Influence Factor)	N	med	sd	N	med	sd	p	δ
Loops vs. Unrolled Loops (Structure)	29	0.50	0.30	29	0.68	0.27	**0.02**	**-0.35**
testPrimitiveTypeClassSerialization	9	0.75	0.20	11	0.75	0.23	0.21	
testReducedFraction	9	0.50	0.23	9	0.50	0.32	0.89	
testContainsIgnoreCase_LocaleIndependence	11	0.25	0.29	9	0.75	0.22	**0.01**	**-0.67**
Try Catch vs. AssertThrows (Assertions)	27	0.26	0.21	28	0.34	0.27	0.31	
test10	9	0.25	0.22	10	0.25	0.30	0.90	
test2	9	0.25	0.20	9	0.50	0.26	**0.04**	**-0.54**
test303	9	0.25	0.24	9	0.25	0.23	0.89	
Variable Reuse (Structure)	28	0.71	0.19	29	0.66	0.21	0.32	
testInvert	9	0.75	0.18	11	0.50	0.23	0.54	
testNegate	9	0.75	0.22	9	0.75	0.12	0.15	
testAbs	10	0.75	0.17	9	0.75	0.22	0.78	
Package Names, If-Structure,.. (Structure)	28	0.10	0.17	28	0.33	0.22	**0.00**	**-0.59**
test551	8	0.00	0.19	11	0.50	0.21	**0.00**	**-0.82**
test0074	9	0.00	0.22	8	0.25	0.21	0.16	
test1113	11	0.00	0.12	9	0.25	0.11	**0.03**	**-0.51**
Remove Comments (Comments)	28	0.71	0.29	28	0.55	0.21	**0.02**	**0.36**
testContainsRange	9	0.75	0.22	10	0.75	0.17	0.43	
testFactory_double	9	1.00	0.26	9	0.50	0.17	0.08	
testWrap_StringIntStringBooleanString	10	0.75	0.36	9	0.25	0.22	0.15	
Loops vs. Parameterized (Parameterized)	45	0.38	0.35	44	0.59	0.29	**0.00**	**-0.34**
testPrimitiveTypeClassSerialization	14	0.75	0.27	15	0.75	0.22	0.85	
testReducedFraction	14	0.25	0.35	14	0.25	0.30	0.77	
testContainsIgnoreCase_LocaleIndependence	17	0.00	0.20	15	0.75	0.28	**0.00**	**-0.84**
Split Up Tests (Dependencies)	45	0.57	0.33	48	0.76	0.20	**0.00**	**-0.33**
testAllNullBooleans	14	0.75	0.35	17	0.75	0.17	0.08	
testSerializeAndParse	15	0.75	0.32	16	0.75	0.21	0.60	
testSetContentObject	16	0.50	0.29	15	0.75	0.21	**0.01**	**-0.50**
Specific Assertion (Assertions)	44	0.39	0.30	48	0.39	0.26	0.93	
testFourElement2	16	0.50	0.26	17	0.25	0.30	0.44	
showsAllStsGaDownloads	14	0.50	0.31	16	0.50	0.20	0.73	
indexedReadAndIndexedWriteMethods	14	0.25	0.32	15	0.50	0.27	0.47	
Unnecessary Try Catch (Dependencies)	47	0.47	0.28	48	0.57	0.31	0.12	
testChoicesWithValidDefaultValue	16	0.75	0.22	17	0.75	0.27	0.90	
testApplyToMovesValuePassedOnShortName...	15	0.25	0.27	16	0.25	0.27	0.90	
testApplyToWithMultipleTypes	16	0.38	0.29	15	0.75	0.31	**0.01**	**-0.53**
Remove Fixture (Fixture, Test Data)	46	0.75	0.28	46	0.78	0.23	0.87	
Student Example 03	16	0.75	0.33	16	0.75	0.27	0.54	
Student Example 02	15	0.75	0.24	15	1.00	0.18	0.17	
Student Example 01	15	1.00	0.23	15	0.75	0.19	0.10	

Loops vs Parameterized (Figure 6.3f). 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.

Split Up (Figure 6.3g). 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) (Figure 6.3h). 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. [41].

Unnecessary Try Catch (Figure 6.3i). 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 (Figure 6.3j). 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.

RQ 3.1 Findings. Do factors discussed in practice show an influence on readability when scientific methods are used? Applying test code best practices is no silver bullet for improved readability. Statistical analysis of the aggregated results show significant differences between original and modified versions in five out of ten modifications. In these cases, with the expected exception to the modification "Remove Comments", the modifications have a positive influence on readability ratings.

6.2.3 RQ 3.2 What criteria do students use for readability ratings?

In the following sections we present findings from the analysis of the participants explanations for their readability ratings. Additionally we also analyse the participants

perception of their own rating behaviour, based on answers from the post-questionnaire.

Overall criteria

To get an understanding on the overall rating criteria of our participants we count the amount of comments assigned to the super category of influence factors shown in Table 6.4. The top three categories (Test names, Structure, Dependencies) account for around 54% of the comments. Around 11% could not be assigned to any category. This representation gives a hint on the importance of the given super categories to the readability ratings.

Table 6.4: Super category of influence factors with amount of assigned comments.

Super Category	**Σ Com**	**%**
Test names	564	22.3%
Structure	442	17.4%
Dependencies	352	13.9%
Context & Comments	270	10.6%
Assertions	157	6.2%
Test data	114	4.5%
DRYness	106	4.2%
Identifier names	104	4.1%
Parameterized test	57	2.2%
Fixtures	47	1.9%
Unnecessary Try-Catch	31	1.2%
Helper methods /classes	17	0.7%
Unassigned comments	273	10.6%
Sum:	2534	100%

For obtaining insights on more concrete criteria Table 6.5 shows the top 15 selection of the amount of comments assigned to sub categories, which make up for 1651 out of 2534 comments (around 65%). Sub categories for unassigned comments are left out, because they add no further value to this analysis. Each sub category is either prefixed with + to indicate a positive influence or with − to indicate a negative influence on the readability rating.

The super categories of the first six entries are more or less in accordance with the ranking of the super categories in the previous Table 6.4. For test names there is either critique or praise on the descriptiveness in 526 comments (around 21%). For dependencies in 256 comments (around 10%) participants criticise that the test should have been split up into multiple test cases, because different behaviours or functionalities are tested in the given test case. The opposite, the positive influence that a test tests only one thing is mentioned in 46 comments (around 2%). Participants praise or criticise the formatting or structure of the test case in 275 comments (around 11%), because they can not recognize any structure (e.g. Arrange Act Assert) in the test cases or line breaks are not intuitive.

Finally, *Missing Context* is criticised in 103 comments (around 4%) in example when a test performs an action, which needs a deeper understanding of the system under test or the test suite.

Table 6.5: Top 15 of sub category of influence factors with amount of assigned comments.

Sub Category	**Super Category**	**Σ Com**	**%**
– Undescriptive test name	Test names	323	12.7%
– Division into several test cases possible	Dependencies	256	10.1%
+ Meaningful test name	Test names	203	8.0%
+ Good formatting / structure visible	Structure	167	6.6%
– Bad formatting / no structure	Structure	108	4.3%
– Missing Context	Context & Comments	103	4.1%
– Undescriptive variable names	Identifier names	72	2.8%
+ Short	Structure	70	2.8%
– No/Useless comments	Context & Comments	62	2.4%
– Variables/Constants preferred	Test data	51	2.0%
– Too many assertions	Dependencies	50	2.0%
– No parameterized tests used	Parameterized test	49	1.9%
+ Tests only one thing	Dependencies	46	1.8%
+ Simple testcase	Structure	46	1.8%
+ Meaningful comments	Context & Comments	45	1.8%
Sum:		1651	65.1%

After the first six entries the ranking of super categories between Table 6.4 and Table 6.5 begin to differ. Identifier names are criticised in 72 comments (around 3%), most prominently when one letter variables or abbreviation-only variables are used. Appreciation for short test cases is also voiced in 70 comments (around 3%). In 62 comments, participants criticise the absence of explanatory comments or the presence of useless comments. This sub category is related to *Missing Context*, but in this category the participants explicitly demanded explanatory comments, while context can also be provided by other measures e.g. ability to quickly look up the implementation of the system under test. The discussion on literals versus variables or constants is also present in the comments. While 51 comments would prefer variables or constants in some test cases, 15 would prefer literals in other test cases. The critique on too many assertions in 50 comments is related to *Division into several test cases possible* because a test with this shortcoming usually contains multiple assertions. However, comments in the sub category *Too many assertions* did not state that the test should be split up. Too avoid over-interpretation of the comments we keep both sub categories. Finally, participants criticise missed opportunities for usage of parameterized test in 49 comments. This is the case when loops are used or a test case tests one behaviour with different input values.

Participants perception

After the participants rated and commented on the test cases, they were asked to elaborate on their general rating criteria and limitations they observed while rating the test cases. We categorised the free text answers similar to the comments on the rating. Table 6.6 shows the result of the categorisation for both questions. In contrast to the previous categorisation with two levels, we only use one level, because there are fewer answers (one answer per participant and question). When comments match to multiple categories they are counted multiple times. Hence the sums of comments exceed the amount of participants and vary across the tables.

Table 6.6: Participants perception after rating readability of test cases.

(a) Criteria for rating readability reported by participants.

Category	Σ Com	%
Naming	49	26.5%
Structure	41	22.2%
Dependencies	19	10.3%
Context & Comments	17	9.2%
Time to understand	15	8.1%
Assertions	12	6.5%
Experience	11	5.9%
DRY-Principle	11	5.9%
Unassigned	10	5.4%
Sum:	185	100%

(b) Limitations and problems observed while rating readability.

Category	Σ Com	%
Naming	39	26.4%
Dependencies	32	21.6%
Context & Comments	25	16.9%
Structure	21	14.2%
DRY-Principle	11	7.4%
Assertions	5	3.4%
Unassigned	15	10.1%
Sum:	148	100%

The top four rating criteria (Naming, Structure, Dependencies and Context & Comments) shown in Table 6.6a fit to the previously discussed rankings and make up for the majority (around 68%) of comments. This indicates that the participants actual rating criteria and their perceived rating criteria after assessing the test cases are in accordance with each other. The time to understand a test case or own experience gathered from courses; best practices or work also play a role for the rating for some participants.

Table 6.6b shows the limitations and main problems reported by the participants while assessing the readability. The ranking of the categories is the same as in Table 6.6a except for *Structure*. The participants used this question primarily to summarise their main points of criticism on the previously rated tests, which reflects their actual rating well.

> *RQ 3.2 Findings. What criteria do students use for readability ratings?* The high level rating criteria used by the participants of the experiment are *Naming*, *Structure*,

Dependencies and *Context & Comments*. On a lower level test naming, testing only one behaviour, ensuring a clear structure and providing enough context on the system under test are criteria mentioned in a majority of comments.

6.2.4 RQ 3.3 What is the impact of developer experience in context of readability?

In order to investigate the impact of experience on the readability ratings, we first map the results of questions related to software development experience discussed in Subsection 6.2.1 to experience levels as shown in Figure 6.4.

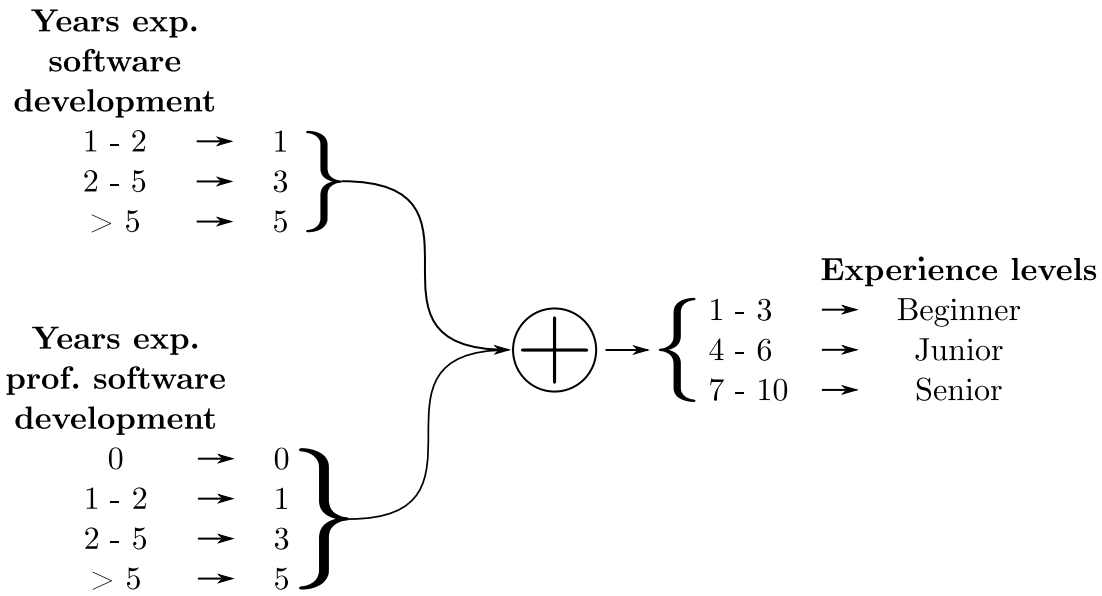
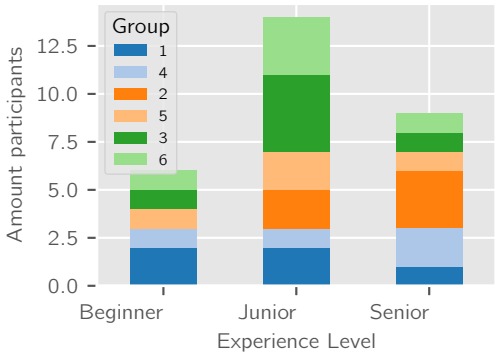


Figure 6.4: Mapping from answers in the questionnaire to experience levels.

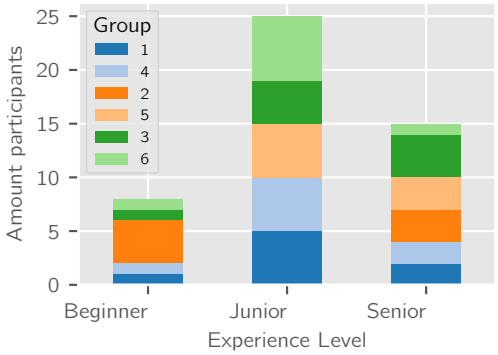
Figure 6.5 shows the resulting distribution of experience levels across both versions of the experiment. In both versions the distribution of experience levels is similar with around 50% of junior- 30% senior- and 20% beginner-level participants. The participants are distributed across the six groups quite uniformly in both versions. Still, not all experience levels contain participants from each group.

Figure 6.6 shows the readability ratings for each experiment version grouped by experience level. The figures show no clear differences in the readability ratings between the experience levels, with exception to the beginner level of the second version in Figure 6.6b. The Wilcoxon rank sum test with a two sided alternative hypothesis shows a similar result when comparing the experience levels beginner vs. junior and junior vs. senior. Only the beginner level of version B shows a significant difference to the junior level with a p-value of 0.0049. The effect size computed with Cliff's Delta gives a value of



Grp	Beg	Jun	Sen	Sum	Perc
1	2	2	1	5	17%
2	0	2	3	5	17%
3	1	4	1	6	21%
4	1	1	2	4	14%
5	1	2	1	4	14%
6	1	3	1	5	17%
Sum	6	14	9	29	100%
Perc	21%	48%	31%	100%	0

(a) Experiment version A.



Grp	Beg	Jun	Sen	Sum	Perc
1	1	5	2	8	17%
2	4	0	3	7	15%
3	1	4	4	9	19%
4	1	5	2	8	17%
5	0	5	3	8	17%
6	1	6	1	8	17%
Sum	8	25	15	48	100%
Perc	17%	52%	31%	100%	0

(b) Experiment version B.

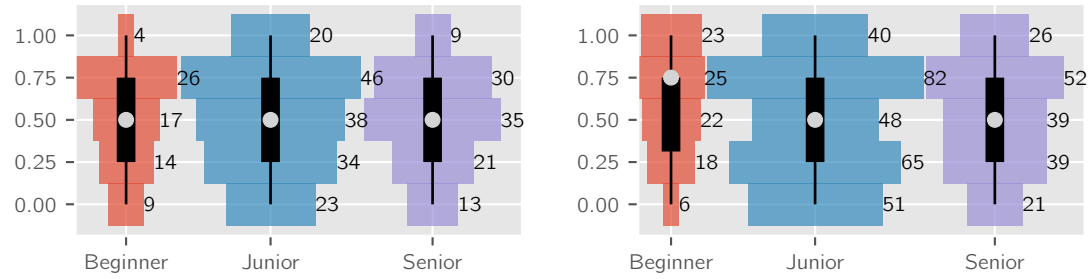
Figure 6.5: Distribution of participants experience levels. For each version groups with similar colors work on the same test cases in reversed order.

0.189 (small effect size). Since the effect size is small and there is no significant difference between the other experience levels, this represents only little evidence for experience influencing readability ratings.

> *RQ 3.3 Findings. What is the impact of developer experience in context of readability?* We found a significant difference with a small effect size in the readability ratings between experience levels in of the four possible cases. Therefore we do not see much impact of developer experience in context of readability.

6.2.5 RQ 3.4 What is the accuracy of automatic readability assessment in comparison to students rating?

In the following section we compare the participants rating with the rating from the readability tool by Scalabrino et al. [64] in order to decide if we should add it to the



(a) Experiment version A.

(b) Experiment version B.

Figure 6.6: Distribution of readability ratings by experience levels.

readability framework. We also report observations from using the tool, which can be used for improving the tool.

Tool description

The readability tool is a command line Java application, which can (amongst others) be used to rate readability of Java code snippets, Java classes or whole projects. When classes or projects are rated, the readability of a class is computed as the mean readability of the methods in the class. The tool including detailed usage instructions is available for download at `https://dibt.unimol.it/report/readability/`. For this comparison we used the latest version of the tool updated in May 2021 [3].

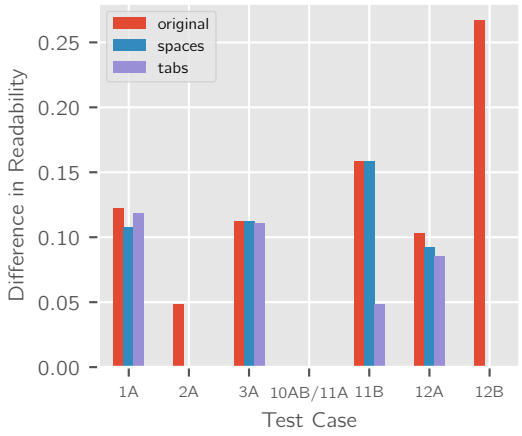
Influence of different whitespace characters

During analysis of the tool generated readability ratings we noticed different results for test cases, which are present in both versions of the experiment. While in most of these cases the different tool ratings can be attributed to minor visual improvements of the test case for the survey form (*Google.Forms* has a rather narrow line width), in two cases there is no difference in the appearance (see Figure 6.7, group 'original', test case 2A, 12B). A close inspection of 2A shows that one line uses space characters for indentation instead of tabs and there are trailing tab characters in one line. In 12B both versions use spaces as indentation but one version contains indented lines without program statements and trailing space characters in one line.

In order to unify this aspect of formatting we adjusted the whitespace characters with a script, which removes trailing whitespace and replaces indentation whitespace with either tabs or spaces (1 tab == 4 spaces). Figure 6.7a also shows the comparison of the adjusted versions (groups 'spaces' and 'tabs'). As expected the difference in the tools rating for test cases 2A and 12B is now 0, because our script removed invisible differences.

[3] SHA1 sum of the download `readability.zip`:
e556b9b05ed14ed76c122170bd7d43fbc39cf80b8acac2930caebe96ac284329

(a) Difference between ratings. The groups denote the whitespace character used for indentation. 'Original' can include tabs and spaces.



(b) Description of differences in formatting.

Test Case	Visible Difference
1A	One line break
2A	No visible difference
3A	Three line breaks
10AB/11A	No difference
11B	Two line breaks
12A	Three line breaks
12B	No visible difference

Figure 6.7: Comparison of tool-generated readability ratings for test cases in both survey versions. Values shown in subfigure a are result of e.g. $|1A_{v1} - 1A_{v2}|$

Since indentation characters and trailing whitespace do not influence the tools rating in the altered versions, the figure shows exemplary for test case 1A, that one line break can alter the tools rating by 0.1 on a scale from 0 to 1.

We further investigate the influence of invisible formatting and apply the previously described script on all test cases. In Figure 6.8 and Figure 6.9 the tools ratings for the different versions are small in most cases, for 56 out of 72 test cases (around 78%) the largest difference is between 0.01 and 0.04. The most extreme difference between the tools ratings appears in test case 3B in Figure 6.8 with a value of around 0.3. Listing 6 shows the corresponding original snippet and its different whitespaces. In the modified versions only one kind of indentation characters are used and trailing whitespaces in the highlighted lines are removed.

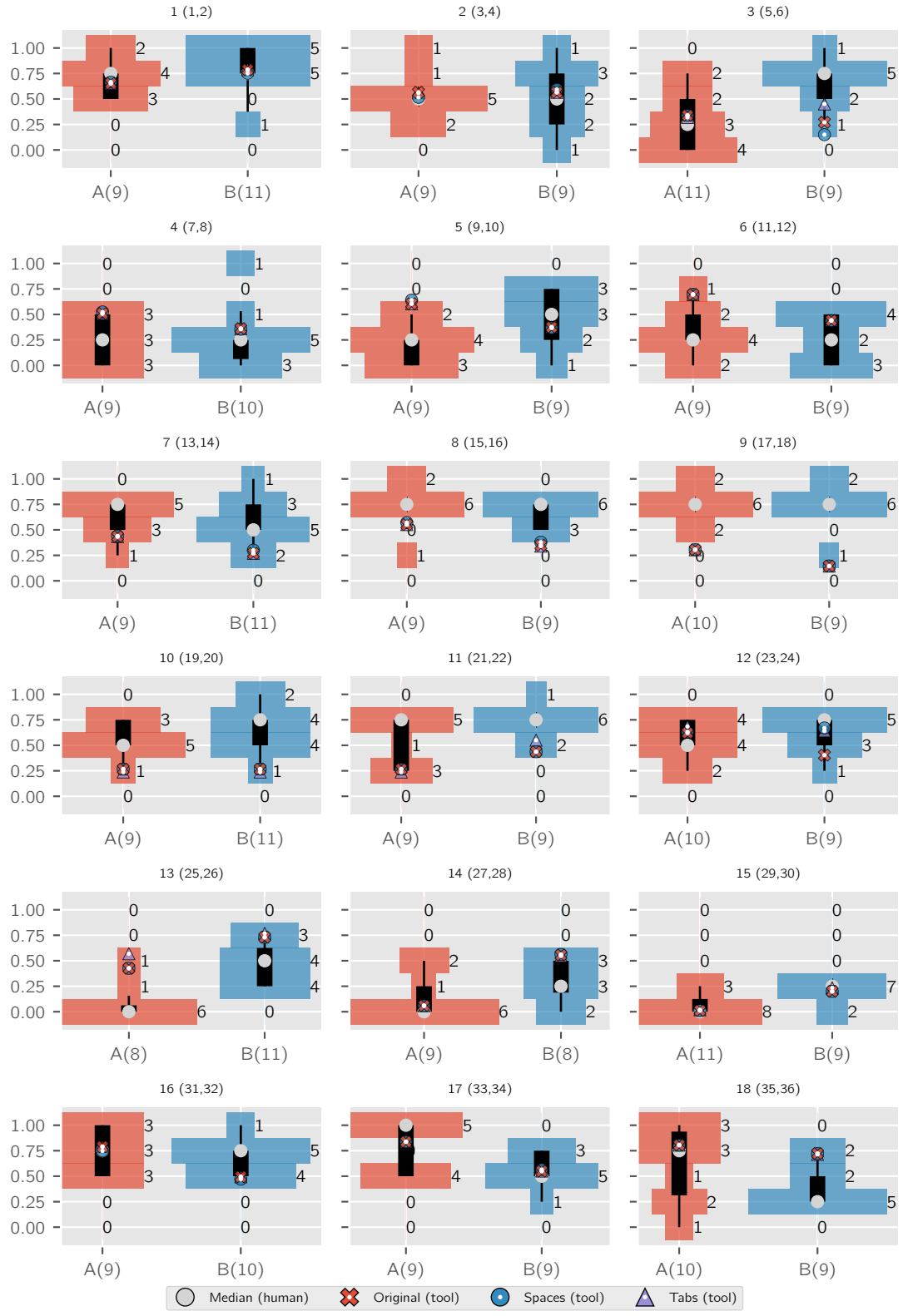


Figure 6.8: Discrete violinplots for participants readability rating and readability tool ratings for survey version A.

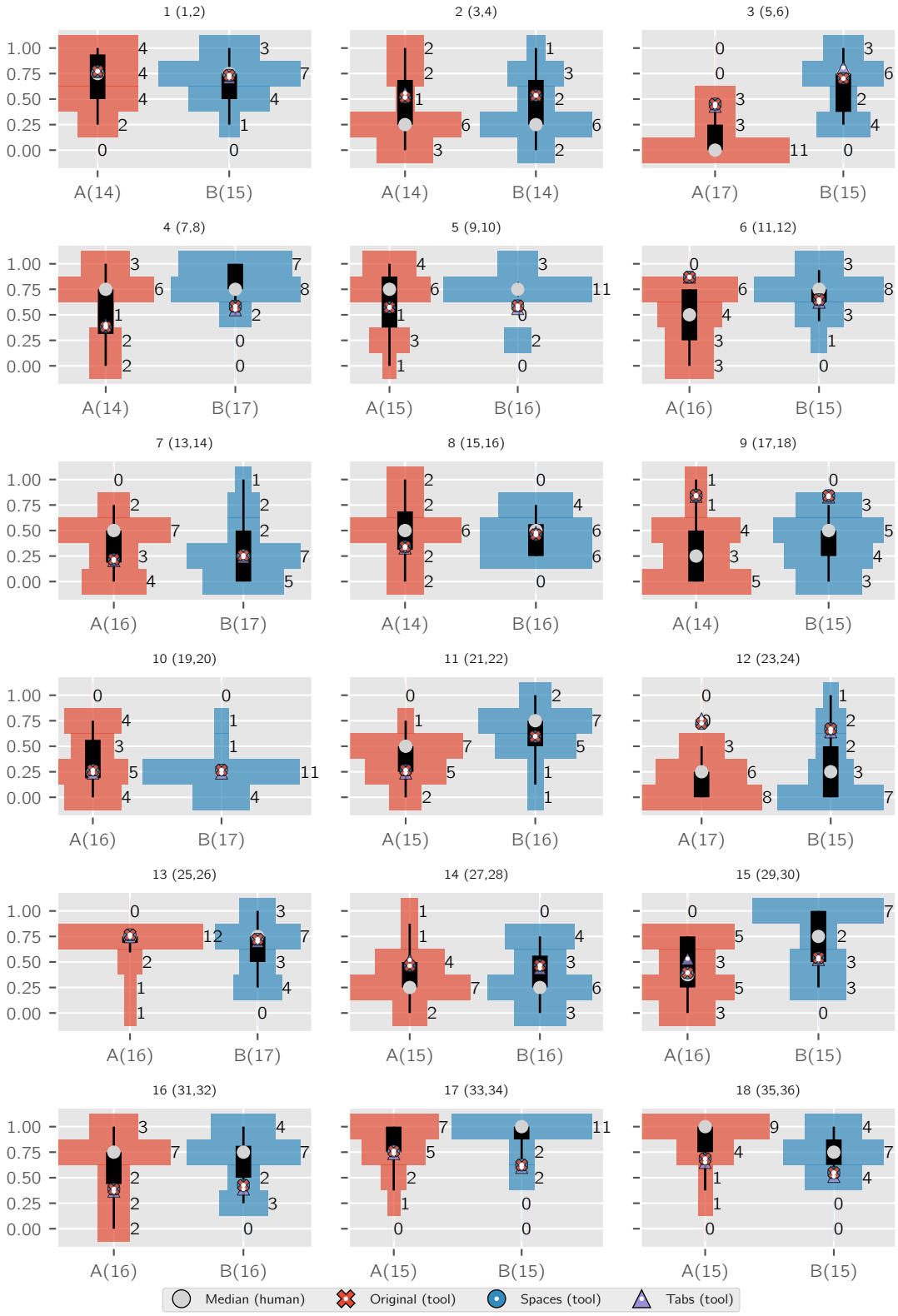


Figure 6.9: Discrete violinplots for participants readability rating and readability tool ratings for survey version B.

```
1  public␣class␣Test␣{
2
3  ␣␣␣␣@DefaultLocale(language␣=␣"de",␣country␣=␣"DE")
4      →@Test
5      →public␣void␣testContainsIgnoreCase_LocaleIndependence()␣{
6      →   →Locale.setDefault(Locale.ENGLISH);
7      →   →assertTrue(StringUtils.containsIgnoreCase("i",␣"I"));
8      →   →assertTrue(StringUtils.containsIgnoreCase("I",␣"i"));
9      →   →assertTrue(StringUtils.containsIgnoreCase("\u03C2",␣"\u03C3"));
10     →   →assertTrue(StringUtils.containsIgnoreCase("\u03A3",␣"\u03C2"));
11     →   →assertTrue(StringUtils.containsIgnoreCase("\u03A3",␣"\u03C3"));
12 ␣␣␣␣␣␣␣␣assertFalse(StringUtils.containsIgnoreCase("\u00DF",␣"SS"));
13
14     →   →Locale.setDefault(new␣Locale("tr"));
15     →   →assertTrue(StringUtils.containsIgnoreCase("i",␣"I"));
16     →   →assertTrue(StringUtils.containsIgnoreCase("I",␣"i"));
17     →   →assertTrue(StringUtils.containsIgnoreCase("\u03C2",␣"\u03C3"));
18     →   →assertTrue(StringUtils.containsIgnoreCase("\u03A3",␣"\u03C2"));
19     →   →assertTrue(StringUtils.containsIgnoreCase("\u03A3",␣"\u03C3"));
20 ␣␣␣␣␣␣␣␣assertFalse(StringUtils.containsIgnoreCase("\u00DF",␣"SS"));
21 ␣␣␣␣␣␣␣␣
22     →   →Locale.setDefault(Locale.getDefault());
23     →   →assertTrue(StringUtils.containsIgnoreCase("i",␣"I"));
24     →   →assertTrue(StringUtils.containsIgnoreCase("I",␣"i"));
25     →   →assertTrue(StringUtils.containsIgnoreCase("\u03C2",␣"\u03C3"));
26     →   →assertTrue(StringUtils.containsIgnoreCase("\u03A3",␣"\u03C2"));
27     →   →assertTrue(StringUtils.containsIgnoreCase("\u03A3",␣"\u03C3"));
28 ␣␣␣␣␣␣␣␣assertFalse(StringUtils.containsIgnoreCase("\u00DF",␣"SS"));
29     →}␣␣
30 }
```

Listing 6: Test case 3B original. Highlighted lines show trailing spaces.

Accuracy of tool ratings compared to human ratings

In the following section we compare the tool generated ratings to the rating data from the experiment. With the previous findings in mind we define that the tool represents the participants rating of a test case iff at least one of the three ratings from the tool is in between the 0.25% and the 0.75% quantile of the test case. We round the tools rating to two decimal points. The violinplots in Figure 6.8 and Figure 6.9 contain boxplots where the boxes visualize the 0.25% and the 0.75% quantile. As an example in Figure 6.8 the tools rating in test case 1B reflects the participants rating, because the *spaces* rating is inside the box.

Table 6.7: Amount of test cases where the tools rating is between the 0.25% and the 0.75% quantile (i.e. the 'box' of the boxplot) of participants ratings.

		Tool Rating	
Exp.	**Version**	**In Box**	**Outside Box**
A		15	21
B		22	14
Sum:		37	35

Table 6.7 shows result of applying the above definition on all test cases. For 37 out of 72 test cases (around 51%) the tools rating is in accordance with the participants ratings.

If we add special cases, where the range of the box is zero and the tools ratings are in vicinity (Version A: 2A, 15B. Version B: 10B), the result would increase to 40 out of 72 tool ratings in accordance with participants ratings (around 56%).

RQ 3.4 Findings. What is the accuracy of automatic readability assessment in comparison to students rating? In 37 out of the 72 test cases (around 51%) the tools rating is between the 0.25% and the 0.75% quantile of participants ratings. Invisible differences in formatting can influence the tools rating in extreme cases up to 0.25 on a scale from 0 to 1.

CHAPTER 7

Readability Framework - Development and Evaluation

We present the readability framework which consists of a set of questions and supplementary information in the following sections. The framework is based on the findings of the literature surveys from Chapter 4 and Chapter 5 and the experiment from Chapter 6. The selection of the presented readability factors is based on the results from the investigation of factors discussed in practice (see Subsection 6.2.2) and rating criteria of the participants (see Subsection 6.2.5). The guidelines are influenced by the extracted readability factors from the literature surveys (see Subsection 4.2.5, Subsection 5.2.1, Subsection 5.2.2).

7.1 Readability Factor Questions

The following questions in Table 7.1 target the factors influencing readability, which showed significant influence in our experiment or which were used as criteria by the participants of the experiment. These questions can be seen as a checklist, which can be quickly used in a test code review as a basis for discussion.

	−	∼	+
1. Has the test a clear structure?			
2. Is the test free of control structures?			
3. Is the naming of and in the test case adequate?			
4. Does the test verify only one specific behaviour?			
5. Does the test provide enough context?			
6. Does the test use assertions appropriately?			

Table 7.1: Set of questions related to readability factors.

Each question has three possible answers, which are assigned to a certain amount of points. Summing up the points gives a readability score for a test case ranging from 0 (not readable) to 12 (well readable). This score allows a comparison with other readability assessments.

− (0 pt.): the test **mostly fails** the criterion.

∼ (1 pt.): the test **fulfils and fails** the criterion in almost **equal parts**.

+ (2 pt.): the test **mostly fulfils** the criterion.

7.2 Readability Factor Guidelines

The following sections explain why the factors targeted by the question are important and gives best practice examples and food for thought for improvements.

7.2.1 Structure

Aim for a clear structure. Empty lines or comments for enforcing structure

Why: A structure, which clearly separates common parts of a test method helps to locate them. It also allows the reader to expect certain actions from these parts like in a newspaper article, which can be divided into header, teaser and main text.

How: There exist various structuring patterns, three popular ones are AAA (*Arrange; Act; Assert*), BOC (*Build; Operate; Check*) and GWT (*Given; When; Then*). These patterns all follow the same approach, the first step sets up the scene e.g. initializes variables and prepares the system under test (SUT). The second step then performs one or more actions on the SUT which are finally checked with assertions in the third step. Empty lines or even comments indicating the current step can be used to emphasize this structure.

Avoid control structures. Use alternatives e.g. unroll loops, split up tests, parameterize

Why: Control structures add a layer of complexity e.g. in a loop one has to keep in mind the loop variable, in an if-structure one has to evaluate the alternative path.

How: If-structures can be avoided by creating one separate test case for each alternative path of the if-structure or by appropriate usage of assertions (instead of `if (condition) fail();` use `assertTrue(condition);`. Loops can be unrolled when the amount of iterations is reasonable e.g. when adding three items into a data structure. Parameterized tests are a good way to replace loops, which are used to iterate through different instances of one test case e.g. in Listing 8.

```java
@Test
public void testToppingRetrivalAfterReadValidRecipes() {
    IceCreamMachine machine = new IceCreamMachine();
    IceCreamRecipes recipes = new IceCreamRecipes();
    recipes.add(new Recipe().name("Choco-Nuts").ice("Chocolate").topping("Hazelnuts"));
    recipes.add(new Recipe().name("Vanilla").ice("Vanilla"));
    recipes.add(new Recipe().name("Banana-Split").ice("Banana").topping("Chocolate Sauce"));
    machine.readRecipes(recipes);
    assertEquals("Hazelnuts", machine.getToppingOf("Choco-Nuts"));
    assertEquals("None", machine.getToppingOf("Vanilla"));
    assertEquals("Chocolate Sauce", machine.getToppingOf("Banana-Split"));
}

@Test
public void testToppingRetrivalAfterReadValidRecipes() {
    // Arrange
    IceCreamMachine machine = new IceCreamMachine();
    IceCreamRecipes recipes = new IceCreamRecipes();

    recipes.add(new Recipe().name("Choco-Nuts").ice("Chocolate").topping("Hazelnuts"));
    recipes.add(new Recipe().name("Vanilla").ice("Vanilla"));
    recipes.add(new Recipe().name("Banana-Split").ice("Banana").topping("Chocolate Sauce"));
    // Act
    machine.readRecipes(recipes);
    // Assert
    assertEquals("Hazelnuts", machine.getToppingOf("Choco-Nuts"));
    assertEquals("None", machine.getToppingOf("Vanilla"));
    assertEquals("Chocolate Sauce", machine.getToppingOf("Banana-Split"));
}
```

Listing 7: Test without and with emphasised structure.

7.2.2 Naming

Consistent naming. Use naming patterns and conventions for tests and variables

Why: When someone has to read a test case e.g. when looking at a test failure, the test name will be one of the first parts which will be read. Ideally the name prepares the reader for the actual content of the test i.e. it provides enough context on the scenario and summarises the primary intention of the test case. Beside test names variable names are another source for context, hence a consciously chosen name is beneficial to readability. Consistency of test and variable naming might not pay of for a single test but with increasing size of the test suite it e.g. allows developers to expect a certain behaviour from certain variables.

How: Consistent naming can be supported by naming patterns e.g. *test_subject_outcome_scenario* or *givenFooWhenBarThenBaz*. There exist numerous alternatives, some also violate common code style conventions e.g. usage of underscores in Java method names. Such violations can be argued as long as the team uses some pattern to keep test names consistent. Test names can include the tested method instead of the tested behaviour. While this approach facilitates finding new test names it contains the risk that test names have to be edited when the name of the tested method changes. Although the length of test method names is not that critical as the length of production method names, it still can get too long e.g. *testApplyToMovesValuePassedOnShortNameToLong-*

```java
// Test from: org.apache.commons.lang3.StringUtilsContainsTest
@DefaultLocale(language = "de", country = "DE")
@Test
public void testContainsIgnoreCase_LocaleIndependence() {
    final Locale[] locales = {Locale.ENGLISH, new Locale("tr"),
            Locale.getDefault()};

    final String[][] tdata = {
            {"i", "I"},
            {"I", "i"},
            {"\u03C2", "\u03C3"}};

    final String[][] fdata = {
            {"\u00DF", "SS"}};

    for (final Locale testLocale : locales) {
        Locale.setDefault(testLocale);
        for (int j = 0; j < tdata.length; j++) {
            assertTrue(StringUtils.containsIgnoreCase(tdata[j][0], tdata[j][1]));
        }
        for (int j = 0; j < fdata.length; j++) {
            assertFalse(StringUtils.containsIgnoreCase(fdata[j][0], fdata[j][1]));
        }
    }}

@ParameterizedTest
@CsvSource({
        "en, true, i, I ",
        "tr, true, i, I ",
        "de, true, i, I ",
        "en, true, I, i ",
        "tr, true, I, i ",
        "de, true, I, i ",
        "en, true, \u03C2, \u03C3 ",
        "tr, true, \u03C2, \u03C3 ",
        "de, true, \u03C2, \u03C3 ",
        "en, false, \u00DF, SS",
        "tr, false, \u00DF, SS",
        "de, false, \u00DF, SS"
})
public void testContainsIgnoreCase_LocaleIndependence(Locale testLocale,
                                                      boolean expected,
                                                      String a,
                                                      String b) {
    Locale.setDefault(testLocale);
    assertEquals(expected, StringUtils.containsIgnoreCase(a, b));
}
```

Listing 8: Test with loops written as a parameterized test.

NameIfLongNameIsUndefined [1]. Besides common naming guidelines, naming certain variables *testee*, *expected*, *actual* or adding these terms to the variable name determines the roles of these variables in the test case.

7.2.3 Dependencies

One test should test one behaviour, multiple assertions can be used

Why: A test, which tests only one behaviour is shorter than a test case which sets up and checks multiple behaviours. This allows developers to get an overview on the

[1] This test was part of the readability rating experiment. In the comments many participants criticised the length of the test name.

test case more quickly. Checking multiple behaviours in one test case can also affect the test name in a negative way. A test name should describe the content of the test in a compact way, but the more behaviours are added to a test case, the more it will become harder to find a compact name. Hence it is likely to end up with a test name like *testClassXWorksAsExpected*.

How: Only asserting properties of the SUT which are strictly necessary for a given test scenario, is a good way to avoid testing multiple behaviours at once. Resist the temptation of asserting additional properties. This goes hand in hand with using as few assertions as possible. For many simple scenarios one assertion is sufficient. When a test case tests different behaviours, splitting up the test into separate tests is the primary option. In special cases e.g. when different combinations of parameters are tested a parameterized test may also be a viable solution.

7.2.4 Context & Comments

Provide enough context with comments or other forms of documentation (assertion messages, documentation of the SUT, etc.)

Why: Abbreviations, complicated test scenarios, quirks of the system can be hard to grasp not only for developers new to a project but also for the initial developers of a project after enough time has passed. Therefore developers have to spend additional time to come up with a sufficient explanation with the risk of misinterpreting parts of the test, which could lead to errors in the future.

How: Although the primary sources for context are test and variable names, values of variables and assertion messages can also provide helpful information. Depending on the project or development environment fast access to the documentation of the SUT also helps with understanding. If the measures before still do not provide enough context, the missing bits of information can be provided by comments. Comments should be seen as a last resort option, because they can clutter the test code and they have to be maintained too.

7.2.5 Assertions

Use appropriate assertions

Why: Testing frameworks provide a wide range of assertion methods intended for checking specific properties with little effort. The intended kind of check is embedded in the name of the assertion, which is valuable information for the reader. One of the best examples is checking exceptions in Java with JUnit5 shown in Listing 9, where the exception handling can be replaced by one assertion.

How: Knowing the testing framework and the assertion it provides is a key requirement for the ability to choose the appropriate assertions. From the point of readability, the

```java
@Test
void testWithInappropriateAssertions() {
    Calculator calculator = new Calculator();
    try {
        calculator.divide(1, 0));
        fail();
    } catch (ArithmeticException exception) {
        assertTrue(exception.getMessage().equals("/ by zero"));
    }
}

@Test
void testWithAppropriateAssertions() {
    Calculator calculator = new Calculator();
    Exception exception = assertThrows(ArithmeticException.class,
        () -> calculator.divide(1, 0));
    assertEquals("/ by zero", exception.getMessage());
}
```

Listing 9: Testing exceptions with inappropriate and appropriate assertions.

kind of assertions e.g. standard JUnit assertions or fluent assertions (e.g. AssertJ) was not found to have an effect on readability.

7.3 Evaluation

In order to evaluate the readability framework, we set up a survey based on the initial human-based readability study presented in Chapter 6 and compare the new results with the initial ratings and the tool ratings.

Setup Survey

In this survey[2] each participant gives ratings to the complete set of test cases shown in Table 7.2[3]. We used this list of 34 original and modified tests [4] in the B-version of the initial readability study in Chapter 6. Since the participants evaluate the questions on the original and the modified version of the same test, we ordered the test cases in a way, that the tests are mixed. That is, tests with the same kind of modification are separated by a minimum of three tests and both versions of one test (the A/B versions) are separated by a minimum of five tests. Additionally, we create a reversed version, to avoid bias coming from the ordering. At the start and at the end of the survey, we again collect information on the participants in a pre-questionnaire and feedback on the survey in a post-questionnaire. Furthermore, at the start of the survey, we provided an overview

[2]Surveys are available under: https://docs.google.com/forms/d/e/1FAIpQLScF_MO3w4fgf7CAzZ9XXH-LxbFLc_hG8fyjR_BOZSwitbNE5g/viewform?usp=sf_link
https://docs.google.com/forms/d/e/1FAIpQLSeJcsMZlJVf851c6pRUnyE7rfh0hZ1uA_AMquxhMZclyJQKvA/viewform?usp=sf_link

[3]In the actual survey, the test of the control group is only rated once. We copied the resulting value in Table 7.2 for symmetry reasons.

[4]Material available under: https://drive.google.com/drive/folders/1pvFgQ4md2_Gpthr_fx6x0H8vTiu89Qly?usp=share_link

on the assessment questions, including short explanations, in order to create a common ground with respect to the meaning and criteria of the questions.

Execute Survey

The survey was open for one week and invitations were sent out to acquaintances of the author and tutors from TU Wien, who all have a background in Java programming. The invited people were asked to start one of the surveys depending on if their day of birthday is even or odd, to allow for an equal distribution across the two groups.

We received a total of five responses, which clearly limits the ability to generalise from the results. We assume that the approximated survey duration of 1h up to 1h 20min and no completion reward was a significant deterrent to potential participants. Hence, we will improve these aspects in future iterations.

Preliminary Results

Although the low response count prevents any sensible statistical analysis, we still take a look at the preliminary results of the ratings generated with the framework. Table 7.2 shows a summary of the aggregated framework ratings from the participants. For this, as described in Section 7.1, each answer gets assigned 0 to 2 points, which are summed up for each participant. This results in a scale from 0 (not readable) to 12 (well readable). In the table, we refrain from normalising the scale to a range from 0 to 1 to avoid decimals. Looking at the standard deviation in 27 cases it is below 2 and in 18 cases the difference between minimum and maximum ratings are lower or equal 3. When we normalise the standard deviation to the 0-1 range in 27 cases the standard deviation is below 0.17. In the initial experiment (see the lower half of tests in Table 6.3), these values are in most cases above 0.20. This suggests, that the ratings from the framework are more concentrated and have less scatter, compared to the initial readability ratings. Nevertheless, there are also cases where the ratings diverge e.g. test cases 4A, 9A have standard deviations from 2.9 to 3.3. Normalised this translates to values from 0.24 to 0.27, which are not uncommon in the initial experiment.

In Figure 7.1, we add the the mean values of the framework ratings to the discrete violinplots of the initial experiment alongside the various ratings of the readability tool. Generally, in 29 of 36 cases, the mean framework ratings rate the tests more readable than the median initial ratings. In some cases e.g. 3A; 9B; 14B, there is a large difference between the frameworks and the median cases. As a detailed example test case 3A contains a nested loop and the frameworks rating is close to the tool ratings but fails to capture the human ratings. Also in most cases (30) the ratings from the framework are higher than the median human ratings. Hence, the framework might profit from adaptations which would lead to lower ratings. For measuring the accuracy of the frameworks ratings we apply the same method as for the tools rating described in Subsection 6.2.5 and used in Table 6.7. That is, we count the amount of test cases where the frameworks rating is in between the 0.25% and the 0.75% quantile of the participants

		A				B			
Idx	Test Name	mean	sd	min	max	mean	sd	min	max
1	testPrimitiveTypeClass Serialization	8.6	1.3	7	10	10.4	2.1	7	12
2	testReducedFraction	6.0	2.0	4	9	8.2	1.3	7	10
3	testContainsIgnoreCase_LocaleIndependence	5.0	1.2	4	7	10.2	1.1	9	11
4	testAllNullBooleans	8.2	3.3	3	12	12.0	0.0	12	12
5	testSerializeAndParse	8.4	0.9	7	9	11.4	1.3	9	12
6	testSetContentObject	8.4	2.5	6	12	10.2	2.5	6	12
7	testFourElement2	5.2	1.9	2	7	5.0	1.6	3	7
8	showsAllStsGaDownloads	8.4	1.8	6	10	7.6	1.1	6	9
9	indexedReadAndIndexedWriteMethods	8.0	2.9	5	12	8.8	1.6	7	11
10	Control	4.6	2.5	2	7	4.6	2.5	2	7
11	Control & Filler	4.6	2.5	2	7	8.4	1.1	7	10
12	Filler	4.4	1.5	3	7	6.0	1.9	4	8
13	testChoicesWithValidDefaultValue	7.4	1.3	6	9	10.6	1.3	9	12
14	testApplyToMovesValuePassedOnShortNameToLongN	7.2	1.5	5	9	10.0	1.6	8	12
15	testApplyToWithMultipleTypes	6.8	1.3	5	8	10.4	1.7	8	12
16	Student Solution03	10.2	0.4	10	11	10.8	0.8	10	12
17	Student Solution02	11.2	0.8	10	12	11.2	1.3	9	12
18	Student Solution01	12.0	0.0	12	12	11.6	0.5	11	12

Table 7.2: Summary of preliminary results of readability ratings generated with the framework. The scale ranges from 0 (not readable) to 12 (well readable).

rating. This range is represented by the 'box' of the boxplots. With this measurement 18 from 36 test cases (50%) are inside the box. The tools ratings shown in Table 6.7 are in the box in 22 cases for these test cases (exp. version B) but in the other version (exp. version A) only 15 cases are in the box.

Taking a look at the working time, the five participants needed something between 1h and 1h30min to complete the assessment of the given 34 test cases (around 2 to 2.5 minutes per test). A direct comparison to the initial experiment, where the participants mostly needed 30 to 60 minutes (around 2.5 to 5 minutes per test) for 12 test cases, is difficult, because in the initial experiment they had to give free text answers. However, in a code review situation, developers will probably have to articulate and reason their opinion on a piece of code. Since the framework provides reasons for the ratings this means that with the help of the framework, developers can faster give reasoned opinions to the readability of test cases.

Summing up, the preliminary results show a lower standard deviation in the ratings compared to the ratings without assessment questions in the initial readability study. The ratings are between the 0.25% and the 0.75% quantile of the initial readability ratings in 50% of the cases. From an efficiency standpoint the rating can help developers to rate and provide reasons to readability of test code faster compared to gut feeling ratings.

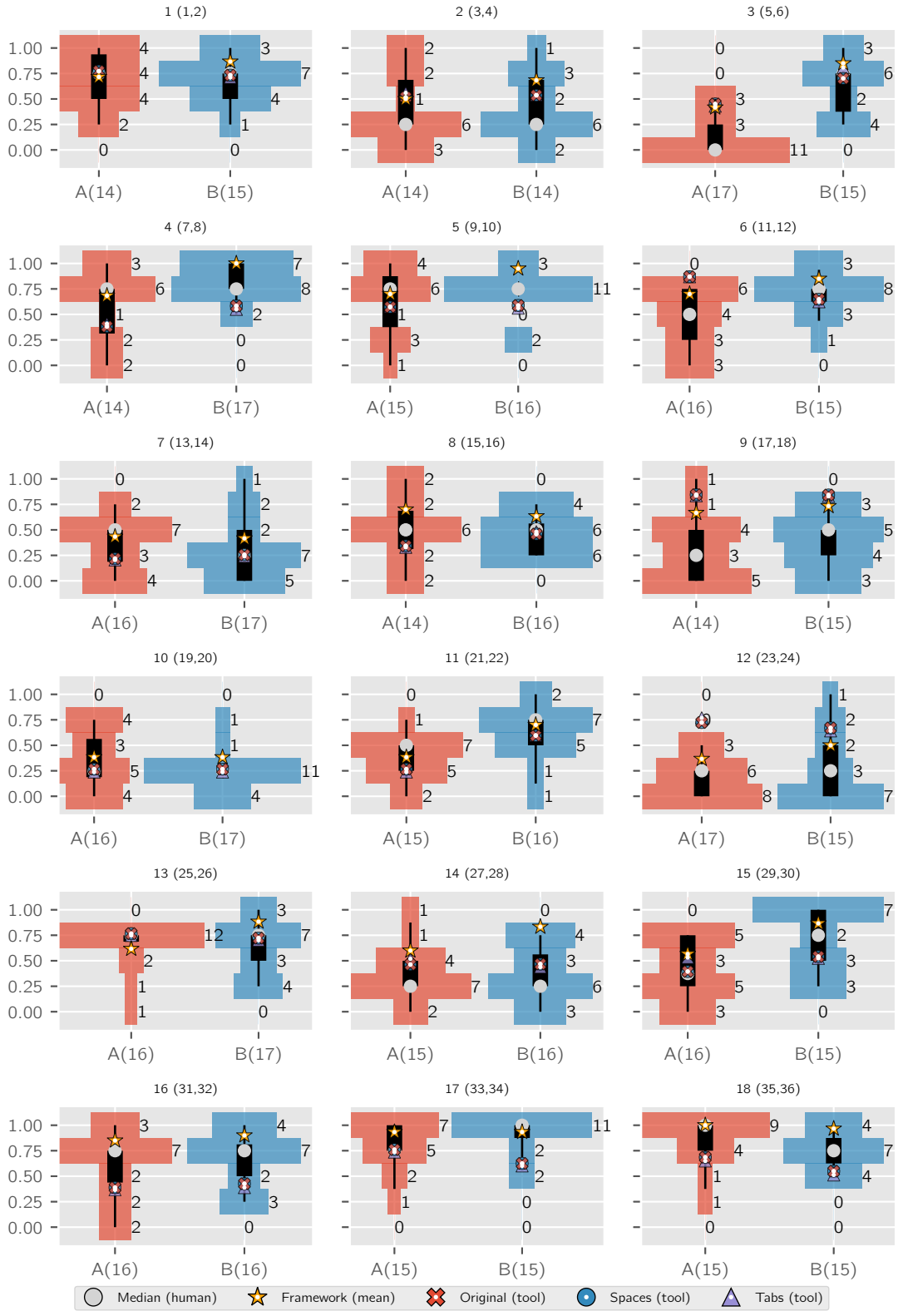


Figure 7.1: Discrete violinplots for initial participants readability ratings, mean readability framework and readability tool ratings for survey version B.

CHAPTER 8

Discussion and Limitations

Firstly we recall and discuss the findings of the research questions we defined to reach the goal of proposing a readability assessment framework for test code. Secondly, we address various threats to validity based on the threats listed in Wohlin et al. [74].

Note: Parts of Section 8.2 are based and extended from a work under review for another publication as Winkler et al. [73].

8.1 Discussion

RQ1: What do we know about test code readability in academic literature?

In the first question we laid the foundation of this work with a systematic mapping study (SMS) with academic literature, with work from Petersen et al. [55, 56] in mind.

RQ1.1: What is the importance of test code readability in scientific communities?
Answer: The results indicate an ongoing general interest in the readability of test code (wide range of venues) with a strong focus on testing related venues and software engineering automation.

Discussion: We did not investigate how the field of test code readability compares to other fields of software engineering with respect to the published studies per year. E.g. it may be that other fields have grown at a faster rate compared to test code readability in the given time period.

RQ1.2: Which types of studies are published and which research methods are used?

Answer: The prevalent types of study are experiments which can contain a survey. Human participation in the experiments is common.

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.

Discussion: Concerning the types of studies and research methods we report the study types reported by the authors of the studies. We rely on the correctness of this information and do not use a different classification scheme.

RQ1.3: Which factors influence readability of test code?

Answer: The majority of studies investigates influence of individual factors on readability. Overall the top three of investigated factors are test names, test structure and identifier names.

Discussion: Although we did not investigate or categorise the magnitude of influence or importance of the influence factor it is interesting to see, that the top two influence factors are in accordance with the top rating criteria obtained from our testing challenge. This confirms the importance of these and further studies in these directions.

RQ1.4: Which kinds of tests were investigated?

Answer: The main focus of readability investigation lies on automatically generated tests, which are often compared to manually written tests.

Discussion: The focus on readability investigation of automatically generated tests comes to no surprise considering that current test code generators have much room for improvement (e.g. Daka et al. [12, 13], Palomba et al. [53], Roy et al. [60]). When improving the readability of test code generators the knowledge on what was improved can also be helpful when humans write test code. By comparing manually written tests to automatically generated tests solely by a readability rating, e.g. a Likert scale, we obtain knowledge about which test is more readable but the factors, which influence this rating, often remain unknown. The approach by Setiani et al. [66] and our approach are different to these studies, because we also extract readability criteria from free text answers.

RQ1.5: Which executable models for assessing code readability exist?

Answer: We found one tool by Scalabrino et al. [64], which can be used for assessing code readability.

Discussion: Although there exist some studies which use executable models to rate readability (e.g. Buse et al. [8], Choi et al. [11], Xu et al. [75]), at the time of writing there only exists this one tool by Scalabrino et al. [64]. However in

the future the work by Karanikiotis et al. [38][1] may be the next state of the art readability rating tool available to the public.

RQ2: What is the opinion of practitioners on test code readability?

We assume developers frequently use the internet in their daily work, for recherche, because it provides easy access to information and inspiration. Hence we systematically gathered sources, which deal with test code readability.

RQ2.1: Which influence factors are discussed in grey literature?

Answer: Most of the sources discuss structural influence factors, test naming, usage of assertions and helper methods and test dependencies.

Discussion: Also a majority of the sources recommend to be consistent, be it in the tests structure when using patterns like *Arrange, Act, Assert*; when naming tests with patterns like *givenFooWhenBarThenBaz*; or naming identifiers.
Overall the recommendations or best-practice examples in the sources seem reasonable, often the authors of the (mostly) blog entries explain their opinion and give code examples. One the one hand there are influence factors where there are pretty clear opinions on the best-practice solution e.g. for *Dependencies* all sources which cover this factor agree that one test should only test one behavior. On the other hand different opinions exist e.g. the inclusion of a concrete method name in the test name. Such differences may be attributed to the features of the programming language, testing framework or generally the context of the author.

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

Answer: There is a clear intersection in the factors investigated by scientific and grey literature. However, the factors are sometimes covered in different ways. We found two influence factors exclusively covered in scientific literature and extracted five new influence factors from grey literature.

Discussion: Overall we found around three times as much relevant grey sources as compared to scientific sources. This is no surprise given the lower entry level for a blog entry compared to a scientific study. The influence factors *Test summaries* and *Textual features* which exclusively appear in academic sources can be argued with the exclusiveness of the underlying technology to academia. That is, test summaries as investigated by the studies are generated from test code and the required software does not seem to be publicly available or well known. This also holds for readability models like textual features. Of course practitioners could also write test summaries themselves but considering that documenting production code and software testing are not the most popular activities, the nonexistent test summaries may not be that surprising.

[1] Readability assessment website by Karanikiotis et al. https://readability-evaluator.netlify.app/

From the grey sources we extracted five new readability factors like usage of *Helper structures* or *Fixtures*. The reason for the absence of these factors in scientific literature may be attributed to the focus of academia on test code generators, which may be not evolved enough to deal with these problems yet. This could also be one reason for the differences in the coverage of readability factors.

RQ3: What insights can we obtain from a readability experiment with students?

We conducted a human-based A/B experiment on readability over multiple years with participants from software testing courses at TU Wien. The participants were asked to rate and reason on test cases from mostly open source software. The test cases target different factors relevant to readability, which we found in the preceding studies on academic and grey literature. We modified the test cases to create the alternate/B versions for the experiment according to best practice. The analysis of the results consists of a statistic analysis with R following an approach, which is also used by a study from Roy et al. [60] with similar data. In order to obtain criteria relevant to readability ratings we categorised and grouped free text answers from the participants.
Last but not least we evaluate the accuracy of readability rating tool from Scalabrino et al. [64] with the ratings obtained from the human experiments.

RQ3.1: Do factors discussed in practice show an influence on readability when scientific methods are used?
Answer: Applying test code best practices is no silver bullet for improved readability. Statistical analysis of the aggregated results show significant differences between original and modified versions in five out of ten modifications. In these cases, with the expected exception to the modification "Remove Comments", the modifications have a positive influence on readability ratings.

Discussion: The analysis confirms that best practices are not hard rules and their successful application depends on the concrete case. The results of *Remove Comments* and *Specific Assertion* are good examples, because for comments a common opinion is to avoid them, because they are an indicator for design flaws. Nevertheless the tests where we removed the comments were rated less readable than the original counterparts. A similar case is the use of assertion with Hamcrest matchers or AssertJ compared to standard JUnit assertions. In the grey sources we found usage of AssertJ or similar assertion libraries is often connected with a more natural language style for asserting properties which suggests that the readability is increased. However, our results show no such effect which is in accordance to Leotta et al. [41]. Another possible explanation for absence of effects could be that the (positive) effects of the modification are masked by more prominent effects which appear in both versions of the test case.

RQ3.2: What criteria do students use for readability ratings?
Answer: The high level rating criteria used by the participants of the experiment

are *Naming*, *Structure*, *Dependencies* and *Context & Comments*. On a lower level test naming, testing only one behaviour, ensuring a clear structure and providing enough context on the system under test are criteria mentioned in a majority of comments.

Discussion: We categorised the free text comments on the participants readability ratings on two categories (super and sub rating criteria, super criteria contain one or more sub criteria). Based on the categorisation or the structure of the groups slightly different results are possible. Still, the top two rating categories of the participants (test naming and structure) are also in the top two of readability factors investigated by scientific and grey literature. Also these results are to some extent comparable to results from Setiani et al. [65].
The rating criteria can be used for justifying further research into other readability factors. Exemplary the for the factor *Dependencies* we only found three publications or for the factor *Structure*, where the publications focus on countable properties of test code more than semantic structure.

RQ3.3: What is the impact of developer experience in context of readability?
Answer: We found a significant difference with a small effect size in the readability ratings between experience levels in one of the four possible cases. Therefore we do not see much impact of developer experience in context of readability.

Discussion: Interestingly the results do not support a strong influence of experience on readability ratings. Indeed in one of the four analysed cases there is a significant difference with a small effect size but this could also be a random deviation. In contrast previous research on maintenance tasks Ceccato et al. [9] or understandability tasks Setiani et al. [65] show an influence of experience on such tasks. Therefore we expected to see clearly visible influence of experience on the readability ratings, because performing such tasks in general includes reading the code at least once. Maybe not overall experience but experience with the concrete system under test is more important.

RQ3.4: What is the accuracy of automatic readability assessment in comparison to students rating?
Answer: In 37 out of the 72 test cases (around 51%) the tools rating is between the 0.25% and the 0.75% quantile of participants ratings. Invisible differences in formatting can influence the tools rating in extreme cases up to 0.25 on a scale from 0 to 1.

Discussion: We evaluated the readability rating tool from Scalabrino et al. [64]. Depending on the definition of accuracy of the tool i.e. 'With which rating does the tool represent the ratings of the humans?', different results can be obtained. Exemplary, if the tools rating only has to be on the side of the rating, where the median rating of students resides, the tool would represent the humans rating in around 66% of the cases (also depending on the special case where the median is

0.5). This may sound promising at first sight but in this setup the possibility for a random correct rating is 50%, because the median can either be in the lower or the upper half of the scale and the tool has a 50:50 chance of 'guessing' the correct half. In combination with the influence of invisible formatting differences e.g. indented lines which are otherwise empty or a mixture of tabulator and space characters for indentation, we do not think that the readability framework would be improved by the addition of this tool yet.

8.2 Limitations

In the following sections we discuss various threats to validity according to Wohlin et al. [74].

8.2.1 Internal validity

Internal validity can be threatened e.g. when the results are affected by a biased selection of participants of an experiment or sources for a literature review. Interactions between participants of an experiment also fall into this category.

- In context of the Systematic Mapping Study (SMS) and the grey literature 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. The SMS was also discussed with external experts.
- The search strings for the SMS focus on the *readability* of test code. The terms *understandability* and *legibility* were used as additional search strings to obtain a wider range of publications.
- The search strings for the grey literature study contain *readability* and *understandability* but not *legibility*, because this search term did not contribute to the result set in the SMS.
- The controlled experiment setup for the initial readability study was tested in a pilot run to ensure consistency of the experiment material. We have used a cross-over design of test case samples to avoid bias of the experiment participants caused by selection and ordering of test cases. The individual runs of the experiment were conducted with different participants, hence learning or saturation effects should be minimal to non-existent.
- Concerning the control groups in the empirical study, the Wilcoxon Rank Sum test does not suggest a significant difference between the readability ratings, when comparing groups with the same questionnaire. However, there is a significant large effect when comparing control groups of different questionnaires. We hypothesize that participants might adapt their rating on the readability of the previous tests.

One of the questionnaires contains six original automatically generated tests. Previous studies have shown that such tests have worse readability than manually written tests. According to our hypothesis the control group in this questionnaire should have a better readability rating than the the control tests in the questionnaire where no automatically generated tests are present. This hypothesis is strengthened by the medians (med_control_gen=0.5 vs. med_control_no_gen=0.25). Additionally, the sign of the large effect size of $\delta = 0.58$ shows that the control group in the questionnaire with automatically generated tests has better readability ratings than other control groups.

Apart from the tests in the control group, the unmodified tests from the modifications *Loops vs. Unrolled* (LU) and *Loops vs. Paramaterized* (LP) are also the same. Hence, we compare the ratings from these A groups. When looking at the median values the hypothesis seems to hold, because the values from the LP modification are lower in two of three tests (med_LU vs. med_LP: 0.75 vs. 0.75; 0.5 vs. 0.25; 0.25 vs. 0). However, the Wilcoxon test does not detect a significant difference in the ratings with $p = 0.11$.

- The participants from the concept evaluation of the readability assessment framework are acquaintances from the author of this work.

8.2.2 External validity

External validity focuses on the overall relevance of the results i.e. how well do the results generalize to day-to-day practice?

- The Systematic Mapping Study is limited to academic context although we include a wide range of publications by searching with Scopus, ACM and IEEE. We reduce this limitation by conducting a survey of grey literature with Google which returned views of practitioners mostly in the form of blogs. By combining the results from both studies we could extract overlapping topics and topics discussed in either academic or practitioner context.
- The experiment was conducted with students from three iterations of the master course *software testing* at TU Vienna, which could limit generalization. By collecting information on the experience of the participants, we know that most participants already worked in industry and have at least junior level experience. Therefore, the experiments results are representative for junior developers.
- Except for three tests, we selected all test cases from real world projects and applied common code refactorings to create alternative versions for the experiment. Nonetheless the results are still limited to the selected test cases.
- The participants primarily rated readability of the test cases with the *google.forms* survey. Since the survey only contains screenshots of the test cases, the appearance of the code (font, highlighting, ...) may differ from their own preferences. To give

the participants the option to use their preferred code appearance, we provided text files with the test cases of the different groups. However, this option was only used rarely.

- The contents of the readability framework originate from the experiment with mostly junior level participants and the literature surveys, which also contains the opinions from practitioners. While this approach is a good foundation, the external validity could be further increased with an evaluation with more participants than the evaluation presented in this work.

- The test cases used in the evaluation of the readability framework are a selection of the tests used in the initial readability study. Hence, mostly test cases from open source software projects where used and the same limitations apply.

- The participants of the framework evaluation only rated the test cases with *google.forms*, hence the same limitations as for the initial readability study apply.

- At this stage the usefulness of the framework for practitioners is limited, because the questions have to be answered manually. We deliberately choose to realise the framework as a questionnaire, because it allows us to evaluate our assumptions quickly. While answering of some of the questions related to syntax, like absence of control structures or appropriate usage of assertions could be implemented with static code analysis tools like *Checkstyle*[2], other questions, which involve the notion of semantics, need a more advanced approach. Such an approach may contain deep learning solutions, which need large amounts of training data. Exemplary, Roy et al. [60] use a dataset of 274 projects containing 96,534 unit test files for the DeppTC-Enhancer in order to improve the readability of unit tests. Since we have not evaluated if the readability factor question can capture human readability ratings at all, such an investment of time is out of scope for this work.

8.2.3 Construct validity

'Construct validity concerns generalizing the result of the experiment to the concept or theory behind the experiment', Wohlin et al. [74].

- We conducted the studies for academic and grey literature with insights and best-practices from literature studies by Garousi et al. [27, 29] and Petersen et al. [56]. The empirical experiment was set up with guidelines proposed by Wohlin et al. [74].

- All investigations in this work are primarily limited to readability, e.g. we did not investigate side effects on maintainability or productivity from the modified test cases used in the experiment. Still, we have the opinion that maintaining readable

[2]Checkstyle homepage: https://checkstyle.sourceforge.io/

tests is in general more enjoyable or pleasant than working with unreadable tests. When the maintenance is more pleasant, there should be less resistance against performing such tasks. Hence, the tests get maintained more often.

- We found no influence of experience on readability ratings in our analysis. However, we do not cover the complete range of experience present in real life. Our participants can have zero to some years of professional software development experience, but we certainly do not have participants with ten or more years of experience. Therefore, this result is also limited to junior-level developers.

- In the experiments announcement and information material the students were told that the reward (bonus points for the course) depends on their active participation. This could motivate the participants to be nit-picky when criticising readability of test cases, because when they find many points to criticise, they can write more text into the comment fields, which proves their active participation. This effect may impact the ability to generalize the ratings for the concrete test cases. Nevertheless, the aggregation of rating criteria is still valid, because the most important points of criticism should still be mentioned the most.

8.2.4 Conclusion validity

The validity of conclusions is threatened when e.g. assumptions for statistical tests are violated or the measures are unreliable which can be the case when a reproduction of an experiment gives different results, Wohlin et al. [74].

- We applied the listed in- and exclusion criteria on the raw results of the literature search and discussed the selected sources as a team. We documented the used search strings, the search engines and the overall process to allow a reproduction of the results.

- The experiment and evaluation survey setup with the online questionnaire in a setting not controlled by the authors introduces the risk of participants being disturbed by random events. We tried to reduce this risk by giving the participants information on the estimated work duration of one run, which allows them to schedule their run.

- The classification of rating criteria was conducted by one of the authors, by assigning comments to a base set of categories and by adding additional categories as needed. Although the author tried to be as objective as possible, it may be that a different author would add other categories or classify comments differently, because a comment is interpreted in another way.

- 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. [60] whose methodology and data is similar to ours.

- Before each statistical analysis with the Wilcoxon Rank Sum test, we tested for normality with the Shapiro-Wilk test. Since the Shapiro-Wilk test does not suggest normality for the data sets and the data is unpaired, we used the Wilcoxon Rank Sum test for every statistical test in this work.
- 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. [30]
- Although the readability framework provides information on the readability factors targeted by the questions, there still is room for interpretation when answering the questions. A software based solution, which would use machine learning approaches, would make the ratings more consistent and to some degree also more objective.
- The concept evaluation of the readability assessment framework has too few participants in order to draw reliable conclusions from the generated ratings. A proper evaluation needs more participants.

CHAPTER 9

Summary and Future Work

In the following sections we summarize our findings, list potential threats to validity and conclude this work with possibilities for future work.

Note: Parts of Section 9.1 are based and extended from a work under review for another publication as Winkler et al. [73].

9.1 Summary

The goal of this work was to propose a framework for readability evaluation of test code. For this we conducted a family of empirical studies and started with a systematic mapping study (SMS) on academic literature to obtain an overview on the influence factors to readability investigated by the scientific community. Furthermore we searched for a tool for rating readability. We broadened the scope of the SMS by adding practitioners perspectives from a grey literature survey. We conducted an experiment in academic context with 77 participants to investigate the impact of a selected set of influence factors from the combination of sources from academia and practice. We also gathered the participants rating criteria and compared the ratings from a readability rating tool to the participants ratings. Finally, we combined the results into a readability framework, which can be used to rate readability of test code and gives information on factors influencing readability of test code.

We have seen an *ongoing general interest in test code readability* in academia based on the timeline of publications and publication venues (see Subsection 4.2.3). An ongoing interest in this topic is also visible from the publication timeline for grey literature sources (see Figure 5.3).

Differences between types of sources and methodology. Scientific literature primarily uses experiments in combination with surveys followed by a statistical analysis for investigating influence factors to code readability (see Subsection 4.2.4). Grey literature mostly consists

of blogs of practitioners who share their own experience or opinion on test code readability (see Figure 5.4).

Unique readability factors. Both grey and scientific literature contain independent sets of readability factors. For scientific literature this set consists of readability models, which combine individual readability factors, and test code summaries (see Subsection 4.2.5). For grey literature we found five additional factors e.g. helper structures or test fixtures. Concerning these factors there exist different views and even conflicting opinions. These can be related to the used/applied technology, testing framework, and test level/approach (see Subsection 5.2.1). The unique readability factors found in grey literature and the overall mapping of readability factors can be used in the scientific community for justifying research into underrepresented topics, which could also improve the readability of test case generators. For practitioners like testers and developers the comparison of influence factors is of interest, because it provides scientific viewpoints on best-practice approaches discussed in grey literature.

Focus of interest. The scientific community has a focus on investigating or improving readability of automatically generated tests (see Subsection 4.2.6). Practitioners on the other side do not deal with with test code generators in the context of readability but focus on problems from their day-to-day work. This may be one of the reasons why certain readability factors are exclusive to the scientific community and practitioners.

Insights from testing experiment. We investigated ten widely discussed readability factors. For five out of ten modifications, which map to readability factors, (Loops vs. Unrolled Loops; Package Names, If-Structures; Remove Comments; Loops vs. Parameterized and; Split Up Tests) there exists a statistical significant influence on the readability of the test cases (see Subsection 6.2.2). The other modifications do not show such a strong influence on the readability, which could be caused by the nature of best practices i.e. best practices are only applicable in certain situations and they are not silver bullets (e.g., modification Try Catch vs. AssertThrows; see Subsection 6.2.2). Our investigation of the participants experience levels shows no clear effect on the readability ratings (see Subsection 6.2.4). This is an interesting result, because previous research found experience influencing maintenance (Ceccato et al. [9]) and code understandability tasks (Setiani et al. [65]). Since such tasks usually require developers to read the code at least once some influence of experience on readability would have been plausible. The comparison of the readability rating tool from Scalabrino et al. [64] to the participants ratings shows that the tools rating does not capture the participants opinion so well that no human inspection is needed (see Subsection 6.2.5). The findings on the influence of different whitespace characters can be used to improve this tool or raise awareness for such influence factors for authors for other tools.

Finally, we extracted *groups of rating criteria* from free text comments into where the most common high level criteria mentioned are *Naming*, *Structure*, *Dependencies* and *Context & Comments* (see Subsection 6.2.3). On that note these results are to some extent comparable to results from Setiani et al. [65]. Just as the collection and mapping of readability factors these rating criteria can be used as input for other research projects,

which e.g. could improve the readability of automatically generated tests. For testers and developers or instructors the criteria can act as information about which readability criteria deserve the most care.

Readability framework. We combined the previous findings in a readability framework, which consists of a questionnaire (see Section 7.1) and guidelines on factors influencing readability (see Section 7.2). While the questionnaire can be used by practitioners for test code reviews, the guidelines provide additional background on individual readability factors and best practice examples. The evaluation with 5 human participants preliminary shows reduced scatter between the participants ratings compared to ratings without the questionnaire. Furthermore, the framework is able to capture the humans rating in half of the test cases investigated, although it often gives a more optimistic rating. That is, the framework rates a test more readable than it might really be. Finally, the participants of the evaluation where more time efficient when assessing the test cases. However, the amount of participants is low, which limits the ability to generalise the evaluation. Nevertheless, the questions can surely act as an input and speed up for test code reviews. Although the ratings generated with the framework may not be accurate in all cases, answering the questions quickly gives a first reasoned impression of the readability. When the developer does not feel like the resulting rating conforms with the gut feeling, she/he can still think about additional reasons, which affect the readability. The collected best practice guidelines give advice on how tests could be improved with respect to readability, which then also influence maintainability positively. These contributions are especially interesting for instructors, developers and testers.

9.2 Future Work

In this work we presented a first proposal for a readability framework and conducted a evaluation with human participants. However, the concept evaluation only consists of 5 participants, which strongly limits its general validity. Therefore, a evaluation on a larger scale would deliver more hints on how the framework could be optimised and provide more robust results with respect to the agreement of raters, the accuracy of ratings and the possible speed up of the review process. Another opportunity for tuning the readability framework is the generation of the readability score. Right now all questions contribute to the rating equally. Since the rating criteria extracted from free text comments are not equally distributed, some factors appear more often than others. Hence, a different weighting of the questions could make the generated score more similar to the aggregated gut feeling readability rating of humans. Additionally, after a thorough evaluation of the framework, machine learning techniques should be utilised to answer the questions, which can not be answered sufficiently by static code analysis tools like Checkstyle. In the current state, the framework generates work for its users when it solely used to generate readability ratings i.e., someone has to answer the questions. Stakeholders like project managers or team leaders would profit from an automated readability assessment, which allows them to get an overview on the quality of their test suites without much additional work.

We also plan to investigate further factors influencing readability and refactorings, which we found with the literature studies, with additional iterations of our testing challenge. This will further deepen our knowledge on test code readability and also lead to improvements of the readability framework.

List of Figures

List of Tables

Bibliography

[1] 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*, pages 352–361, 2013.

[2] M.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*, pages 263–272. Institute of Electrical and Electronics Engineers Inc., 2017.

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[4] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, 2017.

[6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.

[7] 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*, pages 9–14. IEEE Computer Society, 2017.

[8] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2009.

[9] M. Ceccato, A. Marchetto, L. Mariani, C.D. Nguyen, and P. Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *Proceedings - International Conference on Software Engineering*, pages 452–462, 2012. doi: 10.1109/ICSE.2012.6227170.

[10] M. Ceccato, A. Marchetto, L. Mariani, C.D. Nguyen, and P. Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology*, 25(1), 2015. doi: 10.1145/2768829.

[11] Sangchul Choi, Suntae Kim, Jeong-Hyu Lee, JeongAh Kim, and Jae-Young Choi. Measuring the extent of source code readability using regression analysis. In *International Conference on Computational Science and Its Applications*, pages 410–421. Springer, 2018.

[12] 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 Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, pages 107–118, 2015. doi: 10.1145/2786805.2786838.

[13] E. Daka, J.M. 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*, pages 57–67. Association for Computing Machinery, Inc, 2017.

[14] Ermira Daka, José Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. Generating readable unit tests for guava. In *International Symposium on Search Based Software Engineering*, pages 235–241. Springer, 2015.

[15] J. De Bleser, D. Di Nucci, and C. De Roover. Assessing diffusion and perception of test smells in scala projects. In *IEEE International Working Conference on Mining Software Repositories*, volume 2019-May, pages 457–467, 2019. doi: 10.1109/MSR.2019.00072.

[16] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. Socrates: Scala radar for test smells. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, pages 22–26, 2019.

[17] Thomas Deiß. Refactoring and converting a ttcn-2 test suite. *Int. Journal on Software Tools for Technology Transfer*, 10(4):347–352, 2008.

[18] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[19] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[20] Dominic John Farace and Joachim Schöpfel. *Grey literature in library and information studies*. De Gruyter Saur, Berlin ; New York, 2010. ISBN 1282885294. doi: 10.1515/9783598441493.

[21] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–12, 2016.

[22] 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*, volume 2018-January, pages 800–805. Association for Computing Machinery, Inc, 2018.

[23] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[24] 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*, pages 80–89, 2011.

[25] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[26] Erich Gamma, Kent Beck, et al. Junit: A cook's tour. *Java Report*, 4(5):27–38, 1999.

[27] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018. doi: 10.1016/j.jss.2017.12.013.

[28] V. Garousi, B. Kucuk, and M. Felderer. What we know about smells in software test code. *IEEE Software*, 36(3):61–73, 2019. doi: 10.1109/MS.2018.2875843.

[29] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, 2019.

[30] G. Grano, S. Scalabrino, H.C. 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*, pages 348–351. IEEE Computer Society, 2018.

[31] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H.C. Gall. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, 2019. doi: 10.1016/j.jss.2019.07.016.

[32] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C Gall. Pizza versus pinsa: On the perception and measurability of unit test code quality. In *2020 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, pages 336–347. IEEE, 2020.

[33] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[34] David Honfi and Zoltán Micskei. Classifying generated white-box tests: an exploratory study. *Software Quality Journal*, 27(3):1339–1380, 2019.

[35] Paul C Jorgensen. *Software testing: a craftsman's approach*. CRC press, 2018.

[36] Cem Kaner, Jack Falk, and Hung Q Nguyen. *Testing computer software*. John Wiley & Sons, 1999.

[37] Thomas Karanikiotis, Michail D. Papamichail, and Andreas L. Symeonidis. Multilevel readability interpretation against software properties: A data-centric approach. In Marten van Sinderen, Leszek A. Maciaszek, and Hans-Georg Fill, editors, *Software Technologies*, pages 203–226. Springer International Publishing, 2021. ISBN 978-3-030-83007-6.

[38] Thomas Karanikiotis, Michail D. Papamichail, and Andreas L. Symeonidis. Multilevel readability interpretation against software properties: A data-centric approach. In Marten van Sinderen, Leszek A. Maciaszek, and Hans-Georg Fill, editors, *Software Technologies*, pages 203–226. Springer International Publishing, 2021. ISBN 978-3-030-83007-6.

[39] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE, 2009.

[40] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[41] Maurizio Leotta, Maura Cerioli, Dario Olianas, and Filippo Ricca. Fluent vs basic assertions in java: An empirical study. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 184–192, 2018. doi: 10.1109/QUATIC.2018.00036.

[42] B. Li, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and N.A. Kraft. Automatically documenting unit test cases. In *Proceedings - 2016 IEEE Int. Conf. on Software Testing, Verification and Validation, ICST 2016*, pages 341–352. Institute of Electrical and Electronics Engineers Inc., 2016.

[43] B. Lin, C. Nagy, G. Bavota, A. Marcus, and M. Lanza. On the quality of identifiers in test code. In *Proceedings - 19th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2019*, pages 204–215, 2019. doi: 10.1109/SCAM.2019.00031.

[44] 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, pages 204–215. Institute of Electrical and Electronics Engineers Inc., 2019.

[45] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. Ariane 5 flight 501 failure report by the inquiry board, 1996.

[46] Alberto Martín-Martín, Enrique Orduna-Malea, Mike Thelwall, and Emilio Delgado López-Cózar. Google scholar, web of science, and scopus: A systematic comparison of citations in 252 subject categories. *Journal of Informetrics*, 12(4):1160–1177, 2018. ISSN 1751-1577. doi: https://doi.org/10.1016/j.joi.2018.09.002.

[47] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[48] Mathias Meyer. Continuous integration and its tools. *IEEE software*, 31(3):14–16, 2014.

[49] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[50] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 348–359. IEEE, 2020.

[51] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[52] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings - 9th International Workshop on Search-Based Software Testing, SBST 2016*, pages 5–14, 2016. doi: 10.1145/2897010.2897016.

[53] 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*, pages 130–141. Association for Computing Machinery, Inc, 2016.

[54] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H.C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings - Int. Conf. on Software Engineering*, volume 14-22-May-2016, pages 547–558. IEEE Computer Society, 2016.

[55] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*, pages 1–10, 2008.

[56] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[57] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*, pages 73–82, 2011.

[58] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 23–32. IEEE, 2011.

[59] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 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*, volume 177, page 34, 2006.

[60] 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*, pages 287–298. Institute of Electrical and Electronics Engineers Inc., 2020.

[61] Jean E. Sammet. A method of combining algol and cobol. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), page 379–387, New York, NY, USA, 1961. Association for Computing Machinery. ISBN 9781450378727. doi: 10.1145/1460690.1460734.

[62] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th Int. Conf. on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[63] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE, 2017.

[64] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018.

[65] N. Setiani, R. Ferdiana, and R. Hartanto. Test case understandability model. *IEEE Access*, 8:169036–169046, 2020.

[66] 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, volume 2021-August, pages 251–255. IEEE Computer Society, 2021.

[67] S. Shamshiri, J.M. Rojas, J.P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*, pages 250–261, 2018. doi: 10.1109/ICST.2018.00033.

[68] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, Inc., 2014.

[69] Huynh Khanh Vi Tran, Nauman Bin Ali, Jürgen Börstler, and Michael Unterkalmsteiner. Test-case quality–understanding practitioners' perspectives. In *Int. Conf. on Product-Focused Software Process Improvement*, pages 37–52. Springer, 2019.

[70] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*, pages 92–95, 2001.

[71] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[72] Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. 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)*, 2021.

[73] Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. Investigating the readability of test code combining scientific and practical views. Technical Report CDL-SQI 2022-24, CDL-SQI, TU Wien, Vienna, Austria, October 2022. Under review at Empirical Software Engineering Journal (EMSEJ), Special Issue on "Code Legibility, Readability, and Understandability".

[74] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[75] Weifeng Xu, Dianxiang Xu, and Lin Deng. Measurement of source code readability using word concreteness and memory retention of variable names. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 33–38. IEEE, 2017.

[76] 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*, pages 625–636. Association for Computing Machinery, Inc, 2016.

A) Sources of Academic Literature for Systematic Mapping

[A1] 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*, volume 2021-August, pages 251–255. IEEE Computer Society, 2021.

[A2] 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*, pages 287–298. Institute of Electrical and Electronics Engineers Inc., 2020.

[A3] N. Setiani, R. Ferdiana, and R. Hartanto. Test case understandability model. *IEEE Access*, 8:169036–169046, 2020.

[A4] 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*, pages 204–215. Institute of Electrical and Electronics Engineers Inc., 2019.

[A5] G. Grano, S. Scalabrino, H.C. 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*, pages 348–351. IEEE Computer Society, 2018.

[A6] 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*, volume 2018-January, pages 800–805. Association for Computing Machinery, Inc, 2018.

[A7] M.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, pages 263–272. Institute of Electrical and Electronics Engineers Inc., 2017.

[A8] E. Daka, J.M. 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*, pages 57–67. Association for Computing Machinery, Inc, 2017.

[A9] 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*, pages 9–14. IEEE Computer Society, 2017.

[A10] 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*, pages 130–141. Association for Computing Machinery, Inc, 2016.

[A11] B. Li, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and N.A. Kraft. Automatically documenting unit test cases. In *Proceedings - 2016 IEEE Int. Conf. on Software Testing, Verification and Validation, ICST 2016*, pages 341–352. Institute of Electrical and Electronics Engineers Inc., 2016.

[A12] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H.C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings - Int. Conf. on Software Engineering*, volume 14-22-May-2016, pages 547–558. IEEE Computer Society, 2016.

[A13] 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*, pages 625–636. Association for Computing Machinery, Inc, 2016.

[A14] 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*, pages 107–118. Association for Computing Machinery, Inc, 2015.

[A15] 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*, pages 352–361, 2013.

[A16] 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*, pages 80–89, 2011.

B) Sources of Grey Literature for Systematic Mapping

[G1] Gleb Bahmutov. Readable cypress.io tests - gleb bahmutov. Blog, 2019. https://glebbahmutov.com/blog/readable-tests/.

[G2] Anshul Bansal. Best practices for unit testing in java - baeldung. Magazine, 2021. https://www.baeldung.com/java-unit-testing-best-practices.

[G3] Peter Bloomfield. How to write a good unit test - peter bloomfield. Blog, 2020. https://peter.bloomfield.online/how-to-write-a-good-unit-test/.

[G4] Rafał Borowiec. Test code readability improved: Junit with mockito and fest ... Blog, 2013. https://blog.codeleak.pl/2013/07/test-code-readability-improved-junit.html.

[G5] Marcos Brizeno. Write better tests in 5 steps - thoughtworks. Blog, 2014. https://www.thoughtworks.com/insights/blog/write-better-tests-5-steps.

[G6] Vadim Bulavin. Vadim bulavin auf twitter: "9. a good unit test must have three ... Blog, 2020. https://twitter.com/v8tr/status/1217483476406079491?lang=de.

[G7] Dan 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/.

[G8] Roberto Casadei. Effective unit testing - slideshare. Presentation, 2013. https://de.slideshare.net/RobertoCasadei/effective-unit-testing.

[G9] Henry Coles et al. Write damp test code - java for small teams - ncrcoe. Wiki, 2015. https://ncrcoe.gitbooks.io/java-for-small-teams/content/v/restructure/tests/1900_write_damp_test_code.html.

[G10] Amey Dhoke. Do you really want moist test - ameydhoke's blog. Blog, 2009. http://maverick-amey.blogspot.com/2009/05/do-you-really-want-moist-test.html.

[G11] Erik Dietrich. Test readability: Best of all worlds - daedtech. Blog, 2013. https://daedtech.com/test-readability-best-of-all-worlds/.

[G12] Bas Dijkstra. Three practices for creating readable test code. Blog, 2016. https://www.ontestautomation.com/three-practices-for-creating-readable-test-code/.

[G13] Marc Duiker. Improving unit test readability: helper methods & named ... Blog, 2016. https://blog.marcduiker.nl/2016/06/01/improving-unit-test-readability-named-args.html.

[G14] Trevor Foucher Dustin Boswell. *The Art of Readable Code*. O'Reilly, 2012. https://www.oreilly.com/library/view/the-art-of/9781449318482/ch14.html.

[G15] Urs Enzler. Clean code cheat sheet - planetgeek.ch. Cheatsheet, 2014. https://www.planetgeek.ch/wp-content/uploads/2014/11/Clean-Code-V2.4.pdf.

[G16] Javier Fernandes. Rethinking testing through declarative programming. Blog, 2020. https://betterprogramming.pub/rethinking-testing-through-declarative-programming-/335897703bdd.

[G17] Michael Foord. 30 best practices for software development and testing. Magazine, 2017. https://opensource.com/article/17/5/30-best-practices-software-development-and-testing.

[G18] Tobias Goeschel. Writing better tests with junit - codecentric ag blog. Blog, 2016. https://blog.codecentric.de/en/2016/01/writing-better-tests-junit/F.

[G19] Jason Gorman. Readable parameterized tests - codemanship's blog. Blog, 2020. https://codemanship.wordpress.com/2020/09/26/readable-parameterized-tests/.

[G20] Hugh Grigg. A simple, readable, meaningful test style with jest. Blog, 2020. https://notestoself.dev/posts/simple-readable-meaningful-jest-test-style/.

[G21] Philip Hauer. Modern best practices for testing in java - philipp hauer's blog. Blog, 2021. https://phauer.com/2019/modern-best-practices-testing-java/.

[G22] Brian Hnat. Dryer tests - the dumpster fire project. Blog, 2020. https://thedumpsterfireproject.com/dryer-tests.

[G23] Arho Huttunen. How to make your tests readable - arho huttunen. Blog, 2021. https://www.arhohuttunen.com/test-readability/.

[G24] Jason Jarrett. Fluent specification extensions - developing on staxmanade. Blog, 2009. https://staxmanade.com/2009/02/fluent-specification-extensions/.

[G25] Kristopher Johnson. Is duplicated code more tolerable in unit tests? - stack overflow. Stackoverflow, 2008. https://stackoverflow.com/questions/129693/is-duplicated-code-more-tolerable-in-unit-tests.

[G26] Petri Kainulainen. Writing clean tests - petri kainulainen. Blog, 2014. https://www.petrikainulainen.net/writing-clean-tests/.

[G27] Tuomas Kareinen. Readable tests - tuomas kareinen's blog. Blog, 2012. https://tkareine.org/articles/readable-tests.html.

[G28] Vladimir Khorikov. Dry vs damp in unit tests - enterprise craftsmanship. Blog, 2008. https://enterprisecraftsmanship.com/posts/dry-damp-unit-tests/.

[G29] Lasse Koskela. *Effective Unit Testing*. Manning, 2013. https://livebook.manning.com/effective-unit-testing/chapter-4.

[G30] Adit Lal. Kotlin dsl - let's express code in "mini-language" - part 5 of 5. Blog, 2019. https://www.aditlal.dev/kotlin-dsl-part-5/.

[G31] Daniel Lehner. 3 easy fixes for perfect unit test code - devmate. Blog, 2021. https://www.devmate.software/3-easy-fixes-for-perfect-unit-test-code/.

[G32] Daniel Lindner. unit test - schneide blog. Blog, 2013. https://schneide.blog/tag/unit-test/.

[G33] Pawel Lipinski. or how to write tests so that they serve you well. Presentation, 2013. https://2013.jokerconf.com/presentations/03_02_lipinski_pawel_jokerconf-presentation.pdf.

[G34] NAIDELE MANJUNATH and OLIVIER DE MEULDER. No code? no problem — writing tests in plain english - nyt ... Blog, 2019. https://open.nytimes.com/no-code-no-problem-writing-tests-in-plain-english-/537827eaaa6e.

[G35] Robert C. Martin. *Clean Code: Chapter 9*. Pearson, 2021. https://reee3.home.blog/2021/02/17/clean-code-9/.

[G36] Brooklin Myers. Readable test code matters. - brooklin myers. Blog, 2021. https://brooklinmyers.medium.com/readable-test-code-matters-e46cc5c411bb.

[G37] Mark Needham. Tdd: Test dryness - mark needham. Blog, 2009. https://www.markhneedham.com/blog/2009/01/30/tdd-test-dryness/.

[G38] Thomas Papendieck. Why sometimes unit tests do more harm than good? Blog, 2017. https://www.beyondjava.net/why-sometimes-unit-tests-do-more-harm-than-good.

[G39] Corina Pip. Clean code in tests: What, why and how? - testproject. Blog, 2020. https://blog.testproject.io/2020/04/22/clean-code-in-tests-what-why-and-how/.

[G40] Patrick Reagan. Keep your friends close, but your test data closer - viget. Blog, 2009. https://www.viget.com/articles/keep-your-friends-close-but-your-test-data-closer/.

[G41] Jon Reid. 3 reasons why it's important to refactor tests - quality coding. Blog, 2016. https://qualitycoding.org/why-refactor-tests/.

[G42] Jason Roberts. Improve test asserts with shouldly - visual studio magazine. Magazine, 2015. https://visualstudiomagazine.com/articles/2015/08/01/improve-test-asserts-with-shouldly.aspx.

[G43] Jason 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.

[G44] Matheus Rodrigues. What makes good unit test? readability - matheus rodrigues. Blog, 2018. https://matheus.ro/2018/01/15/makes-good-unit-test-readability/.

[G45] Jan Van Ryswyck. Avoid inheritance for test classes - principal it. Blog, 2021. https://principal-it.eu/2021/01/avoid-inheritance-for-test-classes/.

[G46] Anmol Sarna. Do you think your code is perfect? well, think again. Blog, 2018. https://blog.knoldus.com/do-you-think-your-code-is-perfect-well-think-again/.

[G47] Simone Scalabrino. *Automatically Assessing and Improving Code Readability and Understandability.* PhD thesis, Università degli Studi del Molise, 2019. https://iris.unimol.it/retrieve/handle/11695/90885/92359/Tesi_S_Scalabrino.pdf.

[G48] Carlos Schults. Unit testing best practices: 9 to ensure you do it right. Blog, 2021. https://www.testim.io/blog/unit-testing-best-practices/.

[G49] Jenny Shih. A field guide to unit testing: Readability. Blog, 2020. https://codecharms.me/posts/unit-testing-readability.

[G50] John Ferguson Smart. What makes a great test automation framework? - linkedin. Blog, 2020. https://www.linkedin.com/pulse/what-makes-great-test-automation-framework-john-ferguson-/smart.

[G51] Derek Snyder and Erik 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.

[G52] Tengio. More readable tests with kotlin - tengio. Blog, 2016. https://www.tengio.com/blog/more-readable-tests-with-kotlin/.

[G53] Vdaas Vald. The unit test strategy in vald. Blog, 2021. (Company blog) https://vdaas-vald.medium.com/the-unit-test-strategy-in-vald-912ed6f14fbd.

[G54] Vtestcorp. Unit testing tutorial: 5 best practices - vtest blog. Blog, 2020. https://www.vtestcorp.com/blog/unit-testing-best-practices/.

[G55] T. Yonekubo. Readable test code - medium. Blog, 2021. https://medium.com/@t-yonekubo/readable-test-code-cad8a7babc7b.

[G56] Gil Zilberfeld. Test attribute #2: Readability - java code geeks. Blog, 2014. https://www.javacodegeeks.com/2014/07/test-attribute-2-readability.html.