Understanding Test Convention Consistency as a Dimension of Test Quality

Alexa Hernandez

School of Computer Science
McGill University
Montreal, Quebec, Canada

December 2022

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of
Master of Science

Abstract

Unit tests must be readable to help developers understand and debug production code. Most existing test quality metrics assess test code's ability to detect bugs. Few metrics focus on test code's readability. One standard approach to improve readability is the consistent application of conventions. In this thesis, we propose and explore test convention consistency as a dimension of test quality. To that end, we conduct a grey literature survey, eliciting 61 Java JUnit test conventions which we organize into a catalogue of 23 test convention classes. The grey literature suggests convention clashes are likely in team projects, despite multiple developers stressing the importance of consistency. Using the catalogue, we develop Teslo, a prototype tool to measure and visualize the test convention consistency of JUnit test suites. Teslo relies on two novel test convention consistency metrics—one accuracy-based, the other entropy-inspired. Finally, we leverage Teslo to investigate how a developer's participation level relates to the evolution of test convention consistency. We find a significant association between developers with low participation and consistency-degrading commits in 14% of the analyzed test suites. Over 90% of the test suites for which there was no significant association still showed an indication that developers with low participation levels are more likely to degrade consistency.

Resumé

Les tests unitaires doivent être lisibles pour aider les développeurs à comprendre et déboguer le code de production. La plupart des mesures existantes de la qualité des tests évaluent la capacité du code de test à détecter les bogues. Peu de mesures se concentrent sur la lisibilité du code de test. Une approche standard pour améliorer la lisibilité est l'application cohérente des conventions. Dans cette thèse, nous proposons et explorons la cohérence des conventions de test comme dimension de la qualité des tests. À cette fin, nous menons une enquête sur la littérature grise, en obtenant 61 conventions de test Java JUnit que nous organisons dans un catalogue de 23 classes de conventions de test. La littérature grise suggère que les conflits de conventions sont probables dans les projets d'équipe, même si plusieurs développeurs soulignent l'importance de la cohérence. En utilisant le catalogue, nous développons Teslo, un outil prototype pour mesurer et visualiser la cohérence des conventions de test des suites de test JUnit. Teslo s'appuie sur deux nouvelles mesures de cohérence des conventions de test, l'une basée sur l'exactitude, l'autre inspirée par l'entropie. Enfin, nous utilisons Teslo pour étudier comment le niveau de participation d'un développeur est lié à l'évolution de la cohérence des conventions de test. Nous trouvons une association significative entre les développeurs avec une faible participation et des engagements de dégradation de la cohérence dans 14% des suites de test analysées. Plus de 90% des séries d'essais pour lesquelles il n'y avait pas d'association significative montraient encore une indication que les développeurs ayant de faibles niveaux de participation sont plus susceptibles de dégrader l'uniformité.

Acknowledgements

First and foremost, I extend my deepest gratitude to my supervisor Prof. Martin P. Robillard. Without his continuous support and guidance, none of this would have been possible. From ideation to execution, Prof. Martin P. Robillard was involved in each phase, sharing invaluable insights that elevated my research. He went above and beyond the responsibilities of a supervisor, organizing frequent meetups and events, which enhanced my graduate experience. For this, I am sincerely grateful.

I thank the Fonds de recherche du Québec – Nature et technologies (FRQNT) for financial support. Many thanks to my peers in the Software Technology Lab for fostering a warm and collaborative research environment. Our weekly lab meetings were a fantastic source of knowledge sharing, full of productive discussions and useful suggestions. I am grateful to Elby Mackenzie for helping me implement a proof of concept to validate my idea. Special thanks to Mathieu Nassif, who, without fail, was always happy to discuss my research. His thoughtful advice and suggestions helped me refine my research design and gain confidence in my approach.

Finally and most importantly, I am deeply indebted to my family, whose unconditional love and constant encouragement gave me the strength to complete my master's. I am most grateful to Jake for providing me with endless support, patience, and care throughout my master's. I could not have done it without you.

Contents

List of Abbreviations

AAA	Arrange-Act-Assert
ASAT	Automated static analysis tool
AST	Abstract syntax tree
CC	Convention class
DC	Design challenge
GWT	Given-When-Then
IR	Intermediate representation
OR	Odds ratio
RQ	Research question
TCC	Test convention class
TRC	Test-related commit

List of Figures

List of Tables

1 Introduction

Unit tests serve many purposes: they help detect faults, act as documentation, and facilitate debugging activities [1, 2]. The multi-purpose nature of unit tests makes it difficult to define what constitutes a high-quality test. Intuitively, a high-quality unit test should effectively fulfill each of its intended purposes. Devising a metric that captures each of these dimensions is challenging because the factors influencing each dimension do not necessarily align. For example, using descriptive test method names makes tests more effective as documentation and facilitates debugging but does not affect tests' ability to detect faults.

Researchers have proposed various metrics to estimate test quality. Most of these metrics indirectly evaluate test code's ability to detect faults. Of all such metrics, code coverage—the ratio of production code executed by test code—is the most widely researched in prior work and adopted by practitioners. Nevertheless, a recent study revealed that practitioners find code coverage insufficient as a test quality metric [1]. They believe code coverage paints an incomplete picture of test quality as it fails to assess test code's ability to perform two of its three purposes—help developers understand and debug production code.

Measuring whether test code is effective as documentation or facilitates debugging is more complicated. In both cases, fulfillment is impacted not only by the test's design but also by the perception of the developer using that test. Still, for any chance of success, developers must be able to *read* the test code. As a result, past studies show that practitioners deem readability crucial to achieving high-quality tests [1, 3, 4]. These findings align with various unit testing doctrines that include readability in their guiding principles [5, 6].

Despite the perceived importance of readability, test code is significantly less readable than production code in practice [7]. For example, Li et al. [8] found that more than half of the 212 developers surveyed experience “moderate” to “very hard” difficulty understanding unit tests. Likewise, by surveying 225 developers, Daka and Fraser [9] identified difficulty understanding tests as a top obstacle to fixing failing tests.

One widely accepted means of improving code readability is the consistent application of code conventions. As evidence, Oracle’s main argument for adopting its *Coding Conventions for the Java Programming Language* is “[c]ode conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly” [10]. It is not the conventions themselves that improve readability but rather the uniformity they yield that reduces the cognitive load required to understand the code [11].

In this thesis, we propose and explore test convention consistency as one dimension of test quality. To that end, we elicit contemporary test conventions from relevant online blogs and tutorials. We construct a catalogue of 23 test convention classes that group alternative conventions based on our findings. The catalogue can help guide the development of unit test generation and refactoring tools. The grey literature suggests that Java JUnit developers consider test method name conventions essential to test quality. They believe test method name conventions enhance readability and improve productivity by formalizing how to express the intent of a test. At the same time, the results suggest that the probability of convention clashes in team projects is high, with 15 distinct conventions mentioned by only 13 authors. Since convention clashes reduce gains in readability and productivity, multiple authors emphasize that consistency is of utmost importance. The findings thus affirm test convention consistency as a signal for test quality and underscore the need for tools to measure and improve it.

We leverage the catalogue to design and develop Teslo, a novel prototype tool in Java to monitor and visualize the consistency of JUnit test suites. In doing so, we posit two novel metrics to measure test convention consistency, each effective for different use cases. The first metric is accuracy-based and assumes the most frequent convention within a class is the target one. The second metric represents the consistency of a test convention class as the inverse of its entropy. The first metric is prescriptive and easier to interpret, whereas the second is descriptive and more robust against different occurrence distributions. Neither metric requires developers to have a priori knowledge of their team's preferred convention. Hence, developers can leverage Teslo to understand which conventions their team inherently prefers and to what degree. They can then make data-driven decisions about which conventions to adopt team-wide and detect deviations with Teslo. Moreover, we designed Teslo to facilitate customizations and understandability, mitigating two commonly cited barriers to adopting automated static analysis tools [12, 13, 14].

Finally, we use Teslo to explore the evolution of consistency in open-source test suites. Specifically, we conduct a multi-case study of open-source JUnit test suites to determine whether developer participation level relates to the evolution of test suite consistency. We find a significant association between developers with low participation levels and consistency-degrading commits for 11 (14%) of the 80 test suites studied. In those test suites, developers with lower levels of participation have between 1.20 and 2.83 greater odds of contributing consistency-degrading commits. Although not significant, over 90% of the remaining 69 test suites still show a negative relationship between a developer's participation level and the impact of their commits on consistency.

1.1 Motivating Example

We illustrate the problem of test convention consistency by referring to the test suite from the Apache Commons IO library for Java (commit 364f845). It contains three alternative conventions to test exceptional behavior:

1. Using a try-catch block with JUnit's fail method. For example, this convention appears in the following method from the UncheckedAppendableTest class:

```
@Test
public void testAppendCharSequenceIndexedThrows() {
  try {
    appendableBroken.append("a", 0, 1);
    fail("Expected exception not thrown.");
  } catch (final UncheckedIOException e) {
    assertEquals(exception, e.getCause());
  }
}
```

2. Using JUnit's assertThrows method. For example, this convention appears in the following method from the ByteOrderParserTest class:

```
@Test
public void testThrowsException() {
  assertThrows(IllegalArgumentException.class, () -> parseByteOrder("some value"));
}
```

3. Using a helper method. For example, this convention appears in the following method from the FileUtilsCopyDirectoryToDirectoryTestCase class:

```
@Test
public void copyDirectoryToDirectoryThrowsNullPointerExceptionWithCorrectMessageWhenSrcDirIsNull() {
  final File srcDir = null;
  final File destinationDirectory = new File(temporaryFolder, "destinationDirectory");
  destinationDirectory.mkdir();
  assertExceptionTypeAndMessage(srcDir, destinationDirectory, NullPointerException.class, "sourceDir");
}
```

Despite their different forms, each of the three conventions achieves the same goal: to test whether the focal method throws a specific exception given a specific set of inputs. However, the presence of such alternatives hinders readability by introducing unnecessary distractions. Developers must equate the alternative conventions in their minds, making extracting information from familiar patterns harder. Conversely, if only one convention were adopted consistently, it would reduce the cognitive load required by developers to read the unit tests. Consistently adopting conventions reduces the cognitive load by allowing developers to leverage common patterns to quickly identify the intent of the test and the core components (e.g., focal method, expected exception).

1.2 Contributions

With this thesis, we present four contributions in the area of software testing:

1. The concept of test convention consistency as a dimension of test suite quality. Specifically, we introduce the notion of a *test convention class*, which groups alternative test conventions. We also design two metrics to measure the consistency of a test convention class, each of which is preferable in different contexts.

2. A catalogue of test convention classes elicited from the contemporary grey literature that discusses Java or JUnit test conventions. The taxonomy comprises 23 test convention classes and 61 test conventions.
3. The design of a prototype tool that measures the consistency of Java test suites that use the JUnit framework. The tool leverages the aforementioned catalogue and consistency metrics. It presents the consistency scores in a hierarchical user interface that facilitates comprehension and action-taking. Although the current version of the tool focuses on Java and JUnit test conventions, the tool's architecture and guiding principles are transferable to any type of test convention.
4. An empirical assessment of factors that impact the evolution of test convention consistency. More specifically, an investigation into the relationship between a developer's participation level and the impact of their commits on test convention consistency.

1.3 Thesis Organization

The remaining thesis is structured as follows. Chapter 2 defines relevant software testing terminology and formulates the problem of test convention consistency. Chapter 3 details the grey literature survey methodology and presents the resulting catalogue of test convention classes. Chapter 4 describes the design of Teslo, including the elicitation of three design challenges for detecting consistency in test suites. It also introduces two metrics to measure test convention consistency, along with a discussion of their trade-offs. Chapter 5 presents the methodology and results of the consistency evolution multi-case study. Chapter 6 discusses relevant past research. Chapter 7 presents the conclusion and discusses future work directions.

2 Terminology

This chapter defines relevant software testing concepts and formulates the problem of test convention consistency.

2.1 Software Testing

Software testing is the process of validating that software does what it is supposed to do. There are various types of software tests, including unit tests, integration tests, acceptance tests, and performance tests. In this thesis, we focus on unit testing conventions.

Unit tests check that small, independent pieces of code, typically methods but sometimes classes, perform as expected [15]. Many unit testing frameworks exist to simplify writing unit tests, execute the tests automatically, and report the results. The most popular unit testing framework for the Java programming language is JUnit [16] and is thus the focus of this thesis.

Unit tests consist of the following components:

- **Focal method**: the method whose behavior we are testing.
- **Input data**: the input parameters passed to the focal method.
- **Oracle**: the result we expect when executing the focal method.
- **Assertion**: the comparison of the result of executing the focal method with the oracle. JUnit provides multiple built-in assertions as a part of its org.junit.jupiter.api.Assertions

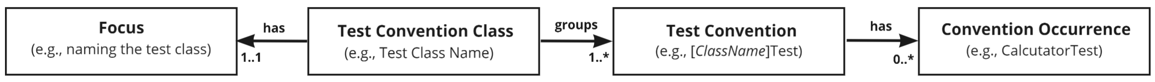


Figure 2.1: Diagram of test convention consistency concepts and their relationships.

class. Each JUnit assertion is a static method that verifies a condition (e.g., assertTrue(boolean condition). JUnit assertions all follow the naming pattern assertXXX and are overloaded to offer flexibility [17]. JUnit reports a test as failed as soon as one of its assertions fails.

As an example, the below unit test validates that the focal method isAdult returns false (oracle) when the input is five using JUnit's assertFalse assertions:

```
@Test
public void test_isAdult_kid() {
  boolean result = isAdult(5);
  assertFalse(result);
}
```

A Java class containing unit tests is called a *test class*. Conventionally, there is one test class per production class that groups all tests for the production class' methods. The collection of test classes within a software project is referred to as a *test suite*.

2.2 Test Convention Consistency

We cast the problem of measuring test convention consistency in the context of a team-based software project that includes a test suite. The test suite contains *test conventions*, some of which may be *alternatives* of each other.

We define a *test convention* as a rule for writing a specific aspect of a unit test. We distinguish between three types of test conventions: code style, programming practices, and programming principles. *Code style* conventions impose rules on the visual appearance of tests, e.g., naming, commenting, and indentation conventions. *Programming practices* define rules on the implementation used to solve a specific task, e.g., a convention to test array equality. *Programming principles* are fundamental laws or truths about unit tests, e.g., unit tests should be independent. Programming principles are not statically checkable.

Moreover, code style and programming practice conventions are typically motivated by a programming principle. For example, developers can use JUnit's setUp and tearDown methods (programming practice) to inject baseline behavior into tests (programming principle). Hence, we exclude programming principles from our definition of a test convention.

A test suite can exhibit *alternative* conventions. We consider two test conventions to be alternatives if they have the same *focus*. Specifically, two code style conventions are alternatives if they impose a rule on the same code element, e.g., two conventions to name the variable storing the oracle. Similarly, two programming practice conventions are alternatives if they define a rule to solve the same task (e.g., two conventions to test array equality). We group alternative conventions into *test convention classes*. Figure 2.1 captures the concepts of test convention, test convention class, and focus, and how they relate to each other.

When two test conventions from the same convention class co-exist in the test suite, an *inconsistency* occurs. Our goal is to measure the test suite's degree of test convention consistency as one dimension of test quality.

3 Identifying Test Convention Classes

Measuring test convention consistency requires a well-defined set of test convention classes. To that end, we sought to answer the research question:

> **RQ1.** *What test conventions are currently employed and discussed by the Java developer community?*

3.1 Method

We surveyed the main content of relevant blog posts and online tutorials. Past work indicates that this kind of *grey literature* provides insight into the state of the practice [18] and is especially valuable for experience- and opinion-based topics [19], such as test conventions [20]. As our goal is to uncover contemporary test conventions used and discussed within the developer community, we target grey literature as the most appropriate source.

We followed Garousi et al.'s [18] guidelines for incorporating grey literature into software engineering research. In particular, we fixed the type of relevant grey literature to blog posts and online tutorial pages. We excluded posts from Question and Answer websites because they target questions with objective and factual answers. For instance, StackOverflow's help center asks users not to post subjective, opinion-based questions [21]. We also excluded classroom materials, e.g., lecture notes, as we are interested in practitioner perspectives.

We used a general search engine, rather than an indexing database as is custom for

Table 3.1: Inclusion criteria for candidate sources in the grey literature survey

	Criteria	Description
Relevance	Content	The source discusses test conventions specific to Java or applicable to unit tests in any language, e.g., naming conventions and commenting conventions.
	Date	The source was published or modified within the last ten years (after September 2011).
Quality	Authority	The author has relevant expertise to form reasonable opinions on test conventions, e,g., they work as a Java developer.
	Validation	The source validates the proposed test conventions, e.g., using code examples [26].
	Impact	The source has at least 20 backlinks. Since practitioners generally reference sources they trust, we use the number of backlinks as a proxy for quality.

white literature surveys, with “java junit test conventions” as our query string. We used the Firefox web browser and, to avoid search bias [22], used the DuckDuckGo search engine. We turned off DuckDuckGo’s localization feature to further ensure search neutrality [23].

We applied inclusion criteria (see Table 3.1), adapted from the work of Garousi et al. [18], to ensure the relevance and quality of the sources. To assess an author’s expertise, we searched for their “about me” section or page on the website hosting the source. If one did not exist, we looked for links to LinkedIn profiles or personal websites and used their content to evaluate the author’s expertise at the publication date. If no such link was present, the source failed the criterion. Consistently with past research [24, 25], we used Ahrefs’ Backlink Checker[1] to count backlinks.

We performed the search on September 3rd, 2021. The first author reviewed the candidate sources in the order displayed by DuckDuckGo. For each candidate source that met the inclusion criteria, the investigator noted all test conventions discussed. The investigator also recorded a description and examples for each test convention discussed. As our goal is to measure test convention consistency as a proxy for readability, we imposed the following

[1]Backlinks are links on other websites that point to the source web page. Ahrefs uses a web crawler to update its backlink index every 15 minutes. https://ahrefs.com/backlink-checker.

inclusion criteria on the identified test conventions:

- **Statically Checkable:** the test convention must be statically checkable to automatically detect its occurrence(s), which is a prerequisite to measuring consistency. An example of a statically checkable test convention is: "Prefix the name of variables storing oracles with *expected*". In contrast, the test conventions "Use the most appropriate assert method" and "Do not rely on indirect testing" are not statically checkable.
- **Generalized:** the test convention must be generalized, that is: (1) the convention may be adopted in any test suite, regardless of the project's domain; (2) if the convention is adopted, it would appear throughout the test suite, not just in a couple of test classes. We focus on generalized test conventions because specialized test conventions are unlikely to impact readability significantly. Examples of generalized test conventions are oracle naming conventions and conventions to handle exceptions. Conversely, examples of specialized conventions include mocking conventions and conventions about testing configuration settings.

We employed a hybrid stopping criterion that blends effort-bounded and saturation-based criteria. Specifically, we analyzed at least 50 sources to ensure adequate coverage. After 50 sources, we stopped when an included source did not yield any new test conventions, or ten consecutive sources failed the inclusion criteria.

After meeting the stopping criterion, the first author used a bottom-up approach to create a catalogue of test convention classes. First, they grouped all alternative conventions into test convention classes. Then, they assigned a name to each convention class based on its focus.

The authors added missing conventions to the catalogue based on their preliminary analysis of open-source test suites and personal experience writing unit tests. To facilitate external validation and ensure transparency, we release the raw data on our online artifact [2]. The raw data comprise all included and excluded sources and test conventions, and the criteria they passed and failed.

[2]https://github.com/AlexaHernandez/TesloResearchArtifact

3.2 Results and Discussion

Thirteen of the 70 sources that we analyzed passed the inclusion criteria. From those 13 sources, we extracted 61 unique test conventions, which we organized into a catalogue of 23 test convention classes. Six identified test convention classes had only one convention. Because a convention class with a single convention can never have inconsistencies, we excluded them from the catalogue presented in Table 3.2. Nevertheless, the complete catalogue is available on our online artifact [3].

Table 3.2: Test convention class catalogue derived from grey literature

Convention Class	Convention
Naming Test Methods	[methodName]_[StateUnderTest]_[ExpectedBehavior] [27, 28, 29, 30, 31, 32, 33, 34]
	[methodName]_[ExpectedBehavior]_[StateUnderTest] [27, 28]
	test[FeatureUnderTest] [27, 28]
	[featureUnderTest] [27, 28]
	should_[ExpectedBehavior]_When_[StateUnderTest] [27, 28]
	when_[StateUnderTest]_Expect_[ExpectedBehavior] [27, 28, 35]
	given_[Preconditions]_When_[StateUnderTest]_Then_[ExpectedBehavior] [27, 28, 35]
	test[MethodName] [36]
	test_[methodName]_with[StateUnderTest]_should[ExpectedBehavior] [29]
	when[Action]Then[ExpectedOutcome] [37]
	given[Preconditions]When[Action]Then[ExpectedOutcome] [37]
	when[StateUnderTest]_Then[ExpectedBehavior] [35]
	[featureUnderTest]_[StateUnderTest]_[ExpectedBehavior] [38, 32]
Naming Test Classes	[ClassName]Test [30, 37, 35, 36]
	[ClassName]Tests [32, 33]
	[ClassName][MethodName]Test [37]
	Test[ClassName]
Naming Oracles	Use the name expected [33, 34]
	Prefix the name with expected [39]
Naming Results	Use the name actual [33, 34]

Continued on next page

[3]https://github.com/AlexaHernandez/TesloResearchArtifact

Table 3.2: Test convention class catalogue derived from grey literature (cont.)

Convention Class	Convention
	Prefix the name with actual [39]
Delimiting Test Sections	Delimit AAA(-T) sections using a comment naming the section and an empty line [31, 33, 34] Delimit AAA(-T) sections using an empty line Delimit GWT sections using a comment naming the section and an empty line Delimit GWT sections using an empty line [32, 39]
Grouping Test Methods	Group related tests using JUnit's @Nested annotation [35] Group the tests for each focal method using JUnit's @Nested annotation [39]
Writing Assertions	Include a message explaining why the test failed [34, 36] Do not include a message explaining why the test failed
Assertion Density	Use one assertion per test method [38, 33, 35, 34] Use a few assertions per test method (2-3) [37] Use many assertions per test method (>3)
Testing Exceptions	Use the expected attribute of JUnit's @Test annotation [38, 33, 35] Use the try-catch idiom with a call to JUnit's fail method in the catch block Use JUnit's assertThrows method
Handling Exceptions	Use a throws Exception statement in the test method's declaration [38] Use a try-catch with a call to JUnit's fail method in the catch block [38]
Repeating Tests Different Input	Use a for loop [31] Use JUnit's parameterized tests [31] Call a helper method that perform the test on different inputs from different test methods
Repeating Tests Same Input	Use JUnit's @RepeatedTest annotation [35] Use a for loop
Setting Up Test Data	Use helper functions [31] Use JUnit's @BeforeEach annotation [35] Use JUnit's @Before annotation [38, 33]
Tearing Down Test Data	Use helper functions [31] Use JUnit's @AfterEach annotation [35] Use JUnit's @After annotation [38, 33]
Scoping Test Methods	Do not test private production methods [31]

Continued on next page

Table 3.2: Test convention class catalogue derived from grey literature (cont.)

Convention Class	Convention
	Test all production methods, regardless of their visibility [38, 35]
Using Static Fields	Do not use static fields [38, 35]
	Use static fields
	Reinitialize static feilds before each test [38, 35]

Readability was the most widely cited reason for adopting test name conventions; for example: "If we want to write tests which are easy to read, we have to stop coding on autopilot and pay attention to naming" [30]. Many authors (54%) explained that test name conventions improve readability by formalizing how they communicate the intent of a test to others, as well as their future selves [29, 30, 31, 38, 32, 37, 35]. In turn, the explicit expression of test intent reduces the cognitive load required to understand tests: "Reading someone else's [test] code is already difficult enough. Any help in understanding it is of much use" [32].

The authors also mentioned increased productivity as motivation for adopting test name conventions. Specifically, various authors argued that proper test names shorten the feedback loop by enabling them to diagnose failing tests [29, 30, 31, 37], understand production code [31, 32, 39], and identify missing tests [37] from test names alone. For instance, one author with more than 20 years of test-driven development experience wrote: "When a unit test fails, I want to understand what I broke—without reading the test code" [38]. With this in mind, another author urged that proper test names are more important than comments because comments do not appear in test runner outputs [37].

Although the authors agreed that test names are of utmost importance, they disagreed on which convention is best. Their disagreement is evident by the sheer number of alternative conventions in the *Naming Test Methods* class. Fifteen distinct conventions were discussed or used by just 13 authors. The first convention is by far the most popular, discussed or used by 62% of authors. It captures the three elements of a test—the method under test, the state under test, and the expected behavior—delimited by underscores. The other alternatives mainly differ in one or more of the following respects: (1) delimitation style (e.g., underscores, camelCamel, mix); (2) filler words (e.g., “test”, “should”, “given”, “a”); (3) inclusion and order of test elements; (4) level of abstraction (e.g., describing the feature under test in natural language instead of using the method name).

The authors provided various rationales for the above alternatives. For example, four authors favored describing the feature under test as it is more robust against name refactorings, lowering the cost of maintenance [27, 28, 38, 32]. One author felt filler words improve readability: “there’s no need to avoid basic English grammar. Articles help the test read flawlessly” [32]. Others argued that they are redundant and lead to long, verbose names [28, 34]. And, while most authors agreed that test names should capture the three elements of a test, one emphasized that the elements need not be explicit, especially when authors share domain knowledge [29]. For example, they compared the explicit yet verbose name test_score_withTwentyZerosRolled_shouldBeZero with the implicit and simple name test_gutterGame.

Overall, the findings highlight the idiosyncratic nature of naming tests, suggesting that convention clashes are likely in team projects. Such convention clashes reduce the gains in productivity achieved when conventions are consistently applied. As two authors emphasized, the key is consistency: “it is important to choose one” [28] and “that everyone on the team knows what conventions are used and is comfortable with them” [37]. These remarks confirm test convention consistency as a signal for test quality and reiterate the need for tools to evaluate and improve test convention consistency.

3.3 Threats to Validity

We may have missed or misclassified some sources or conventions. We mitigated this threat by defining specific and objective inclusion criteria (see Table 3.1). One inclusion criterion for the sources was that they included code examples (i.e., *Validation* criterion). Code examples are easy to identify, reducing the likelihood of the investigator missing or misinterpreting a convention. Nevertheless, we make the raw data, including the excluded sources and conventions, available for external validation.

Although grey literature is the most relevant source of information for our purpose, such publications are not formally reviewed. We mitigated the threat of capturing invalid or unrepresentative test conventions by applying the *Authority* criterion. Using the *Authority* criterion significantly reduced the number of included sources (only 38% of the 70 candidate sources met the criterion). Moreover, we avoided introducing search bias into the candidate sources by using the DuckDuckGo search engine and turning off localization.

4 Measuring Test Convention Consistency

Based on the test convention classes elicited from current grey literature, we aimed to measure the test convention consistency of a test suite as one dimension of test quality. Specifically, we sought to investigate the research question:

RQ2. *How can we automatically measure the test convention consistency of a test suite?*

4.1 Method

We designed and developed an open-source prototype tool called Teslo[1]—short for **TES**ts need **LO**ving too. Teslo is a web application that enables developers to monitor the quality of their test suites. To measure test suite quality, Teslo computes the degree to which test conventions are consistently applied. Teslo measures consistency using two novel metrics described in Section 4.1.1, each of which is preferable in different contexts. Teslo presents the consistency scores at varying granularity levels, from the test suite level down to the test method level.

The closest alternative to Teslo is automated static analysis tools (ASATs) such as FindBugs,[2] PMD,[3] and Checkstyle.[4] ASATs detect common programming errors and conven-

[1]https://github.com/AlexaHernandez/Teslo
[2]http://findbugs.sourceforge.net
[3]https://pmd.github.io
[4]https://checkstyle.sourceforge.io

tion violations using a configurable rule set. Although ASATs can detect issues faster than human inspection, past work has identified several factors hindering their adoption, including poor understandability of tool output [12] and poor customizability [13, 14]. Based on these factors, we sought to overcome three design challenges (DCs) while developing Teslo:

DC1. *Devise consistency metrics that do not require a priori knowledge of the preferred test conventions.*

The default rulesets of ASATs generally focus on deviations from widely accepted conventions. They do not cover the idiosyncratic test conventions we identified from recent grey literature (see Table 3.2). Developers can configure ASATs to detect violations of their team's preferred test conventions. However, doing so requires developers to (1) have a priori knowledge of their team's preferred convention and (2) write custom rules. These requirements are taxing and unrealistic because developers are often unaware of conventions. Code conventions emerge naturally over time as the result of various local decisions made by different developers [40]. Without tool support, it is difficult for developers to synthesize these local decisions into an agreed-upon convention. As such, we aimed to design consistency metrics that developers can use without a priori knowledge of their team's preferred test conventions.

DC2. *Represent test convention classes and test conventions in a way that facilitates customizability.*

ASATs support customizations in terms of enabling and disabling rules and adding custom rules. However, developers find existing tools challenging to configure [12]. They rarely alter the default configurations, adopting ASATs as-is instead [13, 14]. This phenomenon is not due to a lack of demand, as developers deem customizability an essential feature of ASATs [12]. As such, we sought to represent convention classes and conventions so as to facilitate customizability.

DC3. *Display consistency scores in an understandable and actionable manner.*

ASAT reports typically comprise lengthy lists of warnings. Previous work suggests that this output is difficult to understand. For instance, Johnson et al. [12] identified poor understandability of tool output as a significant barrier to use. Fourteen out of 20 developers interviewed mentioned negative impacts due to poorly presented tool output. Software engineers building static analysis tools at Google also found that long lists of warnings seldom motivate developers to resolve all of them [41]. As a result, various studies found that developers only fix a small portion of ASAT warnings [42, 43]. With these findings in mind, we sought to design a user interface that displays consistency scores in a way that facilitates comprehension and action-taking.

4.1.1 Consistency Metrics

We designed two metrics to measure test convention consistency. Our metrics do not require developers to have a priori knowledge of their team's preferred conventions (**DC1**). They allow developers to analyze the alternative conventions their team naturally uses. Using these metrics, developers can make data-driven decisions about which conventions to standardize and enforce. Afterwards, they can also use Teslo to detect deviations.

Accuracy-based Metric

The first metric is accuracy-based. To define an accuracy-based metric, we must identify one "target" or "correct" convention per convention class. We define the target convention to be the convention that occurs the most. The occurrence of any other convention within the convention class constitutes an inconsistency. For example, if a test convention class TCC has three conventions $TC1$, $TC2$, $TC3$, each of which has 50, 25, and 25 occurrences, respectively, then $TC1$ is the target convention. The consistency score of TCC = # consistent occurrences/# total occurrences $= 50/(50 + 25 + 25) = 50\%$.

The accuracy-based metric is prescriptive and useful in cases where developers want to enforce a single convention per convention class. However, the metric is sensitive to the number of occurrences of the target convention. As such, it is not robust against different occurrence distributions. For example, the accuracy-based metric fails to differentiate between the two following test convention classes, each of which has five conventions but different occurrence distributions: $TCC1 = \{50, 15, 15, 10, 10\}$ and $TCC2 =$

$\{50, 49, 1, 0, 0\}$. In both cases, the metric yields an accuracy of 50%. Yet, the latter case is superior in terms of consistency and, consequently, readability. Our second metric mitigates this shortcoming.

Entropy-inspired Metric

The second metric leverages the concept of entropy. Shannon entropy [44] captures the amount of informational value or surprise communicated by a random variable. In our context, a test convention class' entropy (i.e., surprisingness) acts as the inverse of its consistency.

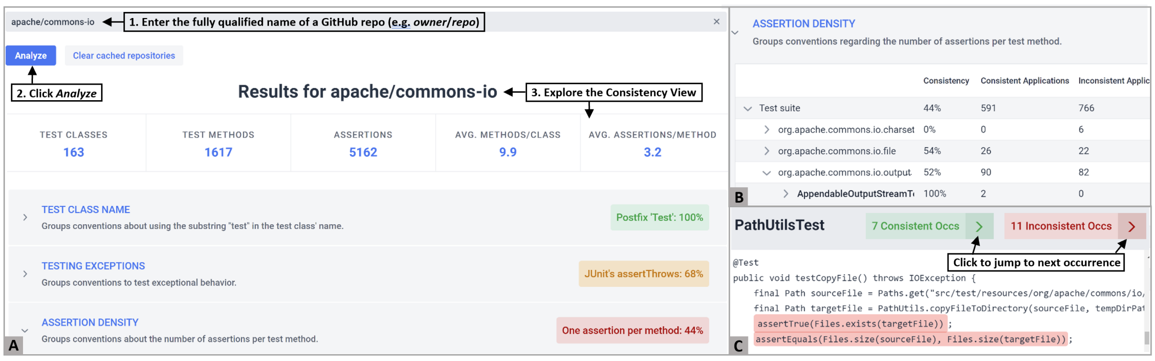
We illustrate this relationship using the test convention class TCC which comprises five conventions. TCC has maximal entropy and minimum consistency when all five conventions occur with equal probability (i.e., 20% of the time). In this case, there is a total lack of uniformity. Developers are unsure of which convention to expect and, thus, must take in more information while reading tests. Reading such tests is difficult as developers must mentally equate five alternative conventions.

Conversely, TCC has minimum entropy and maximum consistency when one of the five conventions occurs 100% of the time. The developer is never surprised as they always encounter the dominant convention. Reading such tests is easier since the familiar convention helps developers quickly decipher the test's intent. Hence, the lower the entropy of a test convention class, the more consistent it is.

Formally, we define the consistency of TCC as

$$\text{Consistency } TCC = \frac{1}{-\sum_{CC} P(CC) log_2 P(CC) + 1} \tag{1}$$

where CC denotes a convention class in TCC. We add one to the entropy in the denominator to avoid a division by zero error when the entropy is zero. The entropy is zero in either of two cases: (1) no conventions within the class occur; (2) only one convention within the class occurs. In either case, there is 100% consistency. Adding one to the de-


apache/commons-io
1. Enter the fully qualified name of a GitHub repo (e.g. *owner/repo*)
Analyze
Clear cached repositories
2. Click *Analyze*
Results for apache/commons-io
3. Explore the Consistency View
A
B
C
Postfix 'Test': 100%
JUnit's assertThrows: 68%
One assertion per method: 44%
PathUtilsTest
7 Consistent Occs
11 Inconsistent Occs
Click to jump to next occurrence

TEST CLASSES	TEST METHODS	ASSERTIONS	AVG. METHODS/CLASS	AVG. ASSERTIONS/METHOD
163	1617	5162	9.9	3.2

TEST CLASS NAME
Groups conventions about using the substring "test" in the test class' name.

TESTING EXCEPTIONS
Groups conventions to test exceptional behavior.

ASSERTION DENSITY
Groups conventions about the number of assertions per test method.

ASSERTION DENSITY
Groups conventions regarding the number of assertions per test method.

	Consistency	Consistent Applications	Inconsistent Applic
Test suite	44%	591	766
org.apache.commons.io.charset	0%	0	6
org.apache.commons.io.file	54%	26	22
org.apache.commons.io.output	52%	90	82
AppendableOutputStreamT	100%	2	0

```
@Test
public void testCopyFile() throws IOException {
    final Path sourceFile = Paths.get("src/test/resources/org/apache/commons/io,
    final Path targetFile = PathUtils.copyFileToDirectory(sourceFile, tempDirPat
    assertTrue(Files.exists(targetFile));
    assertEquals(Files.size(sourceFile), Files.size(targetFile));
```

Figure 4.1: Teslo UI. Frame A (left) illustrates the initial, test-suite-level view of the consistency scores. Frame B (top right) shows how users can drill down to view the consistency scores at the test package, class, or method level. Frame C (bottom right) presents the most granular view—the annotated code dialog that appears when clicking a test class. It allows users to jump through the consistent and inconsistent occurrences in the test class.

nominator yields exactly that (i.e., $1/(0+1) = 1$). Moreover, doing so makes the entropy-inspired metric compatible with the accuracy-based metric by bounding it between zero and one, where a higher score signifies greater consistency.

The entropy-inspired metric is a better proxy for consistency as it places equal weight on the entire occurrence distribution. It is descriptive and valuable in exploratory stages or as an indicator of test quality. Its limitation is that it can be less intuitive. It is also harder to compare consistency scores across convention classes because entropy depends on the number of conventions in the class.

The threat to construct validity for the two proposed metrics is inherently low because the metrics directly abstract a property of the code (consistency between source code elements) as opposed to estimating a subjective human perception (such as readability).

4.2 Teslo Overview

Developers can use Teslo to analyze the consistency of any JUnit test suite that is part of a public GitHub repository.[5] To do so, they enter the fully qualified name of a repository (i.e.,

[5]To analyze test suites that belong to private GitHub repositories, developers can clone the Teslo project or execute Teslo's jar file.

owner/repo) in the input field and click *Analyze*, as depicted in Figure 4.1 Frame A. Teslo then performs a series of data transformations to convert the input repository name into the rendered *Consistency View*. Figure 4.2 illustrates an abstraction of these data transformations and the data that flows between them. First, Teslo takes the input repository name and clones the corresponding repository locally. Next, Teslo parses the cloned repository to generate an intermediate representation (IR) of the test suite (see Section 4.2.1). Then, Teslo analyzes the test suite IR to tally the number of occurrences of each test convention in each test convention class (see Section 4.2.2). Teslo summarizes these occurrence distributions in a *Consistency Report*. Finally, using the *Consistency Report*, Teslo generates the *Consistency View* shown in Figure 4.1 Frame A (see Section 4.2.3). A class diagram of Teslo's design is available on our public GitHub repository.

4.2.1 Parsing Repositories

Teslo parses the cloned repository to generate an IR of its test suite that facilitates test convention detection. To build the IR, Teslo identifies the set of files that belong to the repository's test suite. Teslo considers a file to be a test class if it (1) has the ".java" extension, (2) has "test" in its file path [45, 46], and (3) includes the JUnit import pattern [47] (i.e., import org.junit* for JUnit 4 and JUnit 5 and import junit.framework* for JUnit 3).

For each test class in the test suite, Teslo calls the JavaParser [48] library to generate an abstract syntax tree (AST) of its source code. Teslo traverses the AST to extract all of the test methods in the class. Depending on whether the class has JUnit 3 or JUnit 4+ import statements, Teslo extracts test methods differently. In particular, it extracts all methods that start with "test" and have JUnit's @Test annotation for JUnit 3 and JUnit 4+, respectively.

For each test method, Teslo extracts the list of assertions it contains. Because some convention classes are not JUnit-specific (e.g., Assertion Density), Teslo tries to identify all assertions, not just those provided by JUnit. Although JUnit assertions are sufficient for many testing scenarios, matcher frameworks can help simplify complex testing scenarios. A study of over 4,500 open-source Java projects revealed that developers often use matcher frameworks alongside JUnit [49]. The JUnit team recommends using Hamcrest,[6] AssertJ,[7]

[6]http://hamcrest.org/
[7]https://assertj.github.io/doc/

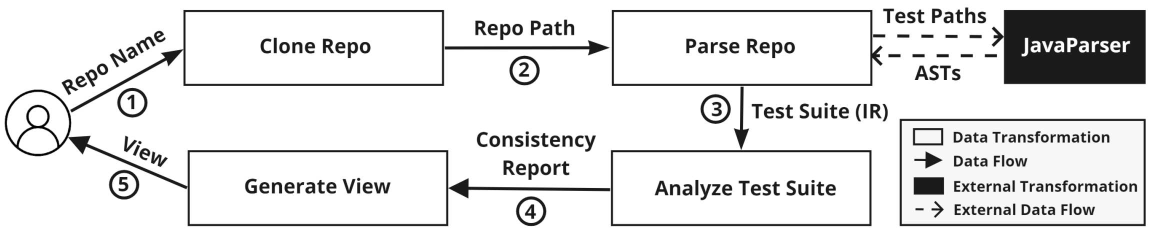


Figure 4.2: Abstraction of the data transformations Teslo performs to convert a repo name into a consistency view.

or Truth,[8] each of which rely on a base method called assertThat. Since JUnit assertions also begin with the substring "assert", we consider any call to a method whose name starts with "assert" to be an assertion. We validated this heuristic by verifying some assertions identified by Teslo. Specifically, we uniformly randomly sampled three assertions per test suite for 30 test suites uniformly randomly sampled from a sampling frame of 928 test suites we elicited for our evolution study (see Chapter 5 for details). All 90 assertions were actual assertions. A total of 59% were JUnit assertions; nevertheless, the other 41%, comprising AssertJ, Hamcrest and custom assertions, would have been overlooked otherwise.

Using the parsed test classes, methods, and assertions, Teslo generates a tree data structure that represents the test suite. The tree is similar to an AST but tailored to test convention detection with nodes representing test packages, classes, methods, and assertions. Each node stores JavaParser AST nodes that are needed to perform test convention detection. For example, if it exists, the test method nodes store a JavadocComment AST node to detect conventions about Javadoc comments.

4.2.2 Analyzing Test Suites

Teslo analyzes the test suite IR to determine how often and where each convention occurs. This analysis relies on a configurable list of *test convention classes*. Each test convention class aggregates one or more test conventions. Test convention objects have one responsibility: to determine whether an input AST node constitutes an occurrence of the convention. Test convention classes also aggregate a convention detector. Convention detectors visit the

[8]https://truth.dev/

test suite IR, identifying the AST nodes that could be an occurrence of a convention in the convention class. For example, the convention detector for Test Class Name extracts all the NameExpr nodes corresponding to test class names. For each candidate AST node, the convention detector calls the isOccurrence method of each convention in the class. This delegation reduces coupling by separating concerns, facilitating customizations (**DC2**). Developers can reuse convention detectors for convention classes with the same candidate AST nodes. Creating new test convention classes is straightforward, as developers can mix and match test conventions and convention detectors. Moreover, we leveraged the Builder design pattern to streamline the creation of test convention instances. For example, defining the convention that test method names should be prefixed with "Test" is as simple as:

```
new Convention.Builder().name("Prefix 'Test'")
                        .description("Prefix test class names with 'Test'.")
                        .isOccurrence(n ->nameAsString(n).startsWith("Test"))
                        .build();
```

Teslo uses the output of the convention detectors to build a *Consistency Report*. The *Consistency Report* is a hierarchical data structure that stores the list of convention occurrences at varying levels of granularity (e.g., test suite level, test package level, etc.). Using this data structure, we can compute the different consistency metrics. The purpose of the *Consistency Report* is to facilitate generating the *Consistency View* that developers see in the user interface.

4.2.3 Generating Consistency Views

After analyzing the repository, Teslo uses the *Consistency Report* to generate a *Consistency View*. We sought to display the consistency scores in an understandable and actionable manner (**DC3**). To that end, and inspired by EclEmma's coverage view,[9] we made the *Consistency View* hierarchical and interactive (see Figure 4.1 Frame A). Specifically, the *Consistency View* starts by showing developers a high-level summary of the test suite's consistency. It consists of an accordion component that has one panel per convention class. Each panel includes a description of the convention class and a badge that reveals the class's dominant convention, i.e., the one that occurs most frequently. The badges also indicate the

[9]https://www.eclemma.org/userdoc/coverageview.html

consistency score of the convention class at the test suite level. The color of the badge varies with the consistency level: green (>= 70%), orange (< 70% and >= 40%), and red (< 40%).

Depending on their information needs, developers can drill down to inspect the consistency scores at the test package, class, and method level. To do so, they expand the accordion panel associated with the desired test convention class (see Figure 4.1 Frame B). This feature allows developers to understand how different parts of the test suite contribute to the overall consistency. Developers can then prioritize fixing test packages and classes with the lowest consistency.

For a more granular view, developers can click on any test class to open a dialog that displays its source code. As demonstrated in Figure 4.1 Frame C, the test class' source code is annotated: the consistent and inconsistent convention occurrences are highlighted in green and red, respectively. Teslo provides buttons to iterate through the consistent and inconsistent occurrences. This dialog helps developers understand the contexts in which they and their teammates apply alternative conventions. Developers can then decide which alternative(s) to adopt team-wide. They can also assess whether deviations from the target convention are justified or unnecessary.

5 Exploring Test Convention Consistency Evolution

Test suites continuously evolve to reflect the changes in the code base they validate. As test suites evolve, so might their consistency. Each change to a test suite enhances, degrades, or maintains its consistency. Understanding which factors impact test suite consistency over time can help developers adopt more proactive approaches to test quality assurance. As a first step, we sought to assess developer participation level as a factor that may impact the evolution test suite consistency. Specifically, we aimed to determine if there exists a relationship between a developer's level of participation and whether their changes degrade test suite consistency. The research questions we sought to answer were:

RQ3.1 *Do peripheral developers degrade test suite consistency more than core developers?*

RQ3.2 *Do new test suite developers degrade test suite consistency more than existing test suite developers?*

5.1 Method

We conducted a multi-case study of open-source Java JUnit test suites hosted on GitHub. Our goal was to study the relationship between a developer's level of participation and whether their commits degrade consistency using Chi-square tests of independence.

5.1.1 Subjects

Our target population is *active* and *collaborative* Java test suites hosted on GitHub that use the JUnit framework. We queried the GitHub API[1] to retrieve a sampling frame. Although we are interested in test suites, the GitHub APIs do not implement such a concept. Instead, we used the APIs to identify a set of candidate Java repositories from which we could extract test suites. GitHub hosts over 10,000,000 public Java repositories (as of 2022-02-14), most of which are not relevant to our study as they are personal projects and inactive [50]. To ensure that the candidate repositories are appropriate, we implemented Kalliamvakou et al.'s [50] GitHub mining peril avoidance strategies in the form of the following inclusion criteria:

- **Active**: the repository must have at least one commit in the past six months (since 2021-08-14) as we are interested in how consistency evolves in *contemporary* test suites.
- **Mature**: the repository must be mature to provide a sufficient amount of test-related commits to analyze. We adopt Jarczyk et al.'s [51] definition of a mature repository: the repository is at least two years old (created before 2020-02-14) and has 100 or more commits on the default branch.
- **Collaborative**: the repository must have at least three unique human contributors since we cast the problem of test convention consistency in the context of a team-based software project.
- **Popular**: the repository has more than 100 stars.

We performed the query on February 14th, 2022, and it produced 4730 candidate repositories.

Next, we constructed our sampling frame by extracting test suites from the candidate repositories. We defined each candidate repository's test suite as the collection of its test classes. We used the same three criteria as Teslo to determine whether a file is a test class (see Section 4.2.1). We only included candidate repositories whose test suite had at least

[1]https://docs.github.com/en/rest

Table 5.1: Summary statistics of the test suites analyzed

	Median	**Min**	**Q1**	**Q2**	**Q3**	**Max**
Test contributors	93	12	67	92	187	960
Test-related commits	1142	237	684	1139	1937	9027
Test classes	243	100	146	237	421	2710
Test suite age (months)	114	39	79	114	153	299
Stargazers	1807	114	164	1803	8195	28552

100 test classes to ensure they had sufficient test-related commits and contributors to analyze. The final sampling frame comprised 928 test suites. We uniformly sampled 80 test suites from the sampling frame for detailed analysis. Table 5.1 captures summary statistics of the 80 sampled test suites.

5.1.2 Unit of Analysis

Test suites evolve through a series of commits. Each commit enhances, degrades, or maintains the test suite's consistency. As such, we chose a *test-related commit* as our unit of analysis. We define a test-related commit as any commit that modifies at least one test file. To identify the set of test-related commits for a test suite, we use git's log command to find all commits that modified at least one file whose path matches the pattern "*test*.java".

5.1.3 Changes in Consistency

We leveraged Teslo to compute the consistency of the sampled test suites after each test-related commit. In this study, we adopted the entropy-inspired consistency metric presented in Section 4.1.1 because it is more descriptive and allows for historical comparisons. Additionally, the entropy-inspired metric more accurately captures the consistency with respect to all alternative conventions, not just a target one.

As described in Chapter 4, Telso computes one consistency score per convention class. We averaged the convention class consistency scores to make the study tractable, yielding a single test-suite-level consistency score. Thus, for each test suite, we generated a series of (test-related commit (TRC), test suite consistency (TSC)) tuples, e.g., (TRC_1, TSC_1), (TRC_2, TSC_2), ..., (TRC_n, TSC_n).

We determined the consistency change generated by a test-related commit by computing the difference between the consistency of the test suite after the current and previous test-related commits. For example, the consistency change (ΔC_2) generated by TRC_2 is equal to $TSC_2 - TSC_1$. We used these consistency changes to classify each test-related commit as consistency degrading ($\Delta C < 0$) or non-degrading ($\Delta C >= 0$).

5.1.4 Developer Participation Level

We employed two different operationalizations of developer participation level, one for each research question. For **RQ3.1**, we used the standard core-peripheral classification discussed in various academic works on open source development [52, 53, 54]. We relied on Mockus et al. [53]'s definition of core developers: the group of the most productive developers who make up at least 80% of the total contributions. Hence, we considered a developer to be core if, at the time of the test-related commit, they were in the group of developers with the highest commit counts that contributed to 80% of the total commit count. Otherwise, we classified the developer as peripheral.

Although the core-peripheral classification captures a developer's level of participation in the overall project, it may not accurately capture a developer's familiarity with the test suite. A developer may contribute significantly to the production code without ever writing supporting unit tests. To account for this discrepancy, we employed a test-specific operationalization of developer participation level for **RQ3.2**. The test-specific operationalization classifies developers as new or existing test suite developers depending on their prior test-related commits. Namely, we considered a developer to be a new test suite developer if, at the time of the test-related commit, they had authored three or fewer prior test-related commits.

5.1.5 Author Identity Resolution

Developers can use different emails and variations of their names when authoring commits. For example, one developer from the stanfordnlp/CoreNLP project[2] authored commits using five distinct combinations of their name and emails. As suggested by Kalliamvakou et

[2]https://github.com/stanfordnlp/CoreNLP

al. [55], we performed name and email unification to classify developers' participation levels more accurately. We considered two (name, email) combinations to belong to the same author if they had the same email or name. However, we only matched on the author's name if the name: (1) was not "no author" or "unknown"; (2) consisted of two or more words (e.g., Jane Smith). We applied this heuristic to reduce false positives since git allows developers to use anything as their name ("bob <bob@gmail.com>") [55]. Our approach correctly unifies all five (name, email) combinations used by the aforesaid stanfordnlp/-CoreNLP developer.

5.1.6 Chi-square Test of Independence

We performed two chi-squared tests for each of the 80 sampled test suites. One to determine whether there exists a relationship between peripheral developers and consistency degradation (**RQ3.1**), and another to determine whether there exists a relationship between new test suite developers and consistency degradation (**RQ3.1**). We used an alpha level of 0.05 to identify likely potential relations between the type of contributor and their impact on the test suite consistency. We applied the Holm-Bonferroni correction [56] to the p-values to account for the effect of multiple comparisons in the probability of observing a significant association. We opted for the Holm-Bonferroni correction because it reduces the probability of type I errors with a lower increase of type II error risk than the Bonferroni correction.

5.2 Results and Discussion

The results reveal that commits made by developers with lower participation levels are more likely to degrade test suite consistency, at least for some projects. Even in the project for which there is no evidence of a significant association, the observed relative frequencies still suggest that developers with lower participation levels contribute consistency-degrading commits more often.

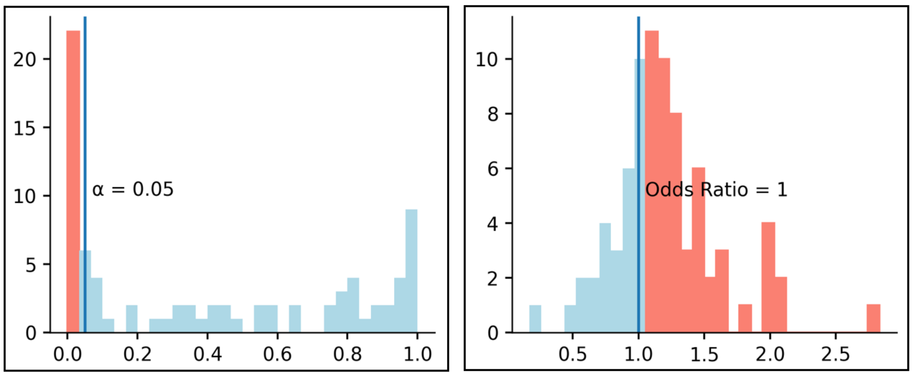


Figure 5.1: Distribution of p-values and odd ratios for the Peripheral Developers vs. Degrade Consistency chi-squared tests

Table 5.2: Peripheral Developer vs Degrade Consistency Chi-square test results (**RQ3.1**): the test-related commits (TRCs), observed counts, p-value, and odds ratio (OR) of the projects which reject the null hypothesis after applying the Holm-Bonferroni correction.

Project (owner/repo)	TRCs	Peripheral/Degrade	Core/Degrade	p-value	OR
alibaba/fastjson	2071	293 (43%)	291 (21%)	2.41e-24	2.83
apache/logging-log4j2	3416	335 (23%)	309 (16%)	4.56e-07	1.60
elki-project/elki	1181	163 (27%)	94 (16%)	2.15e-5	1.95
junit-team/junit5	2772	310 (29%)	367 (22%)	2.95e-4	1.43
netty/netty	3128	629 (32%)	277 (24%)	1.78e-5	1.48
spring-projects/spring-batch	2869	464 (31%)	259 (19%)	3.39e-14	2.00
stanfordnlp/CoreNLP	6144	759 (38%)	1245 (30%)	1.92e-10	1.47
VoltDB/voltdb	9027	1218 (31%)	1377 (27%)	5.97e-4	1.20

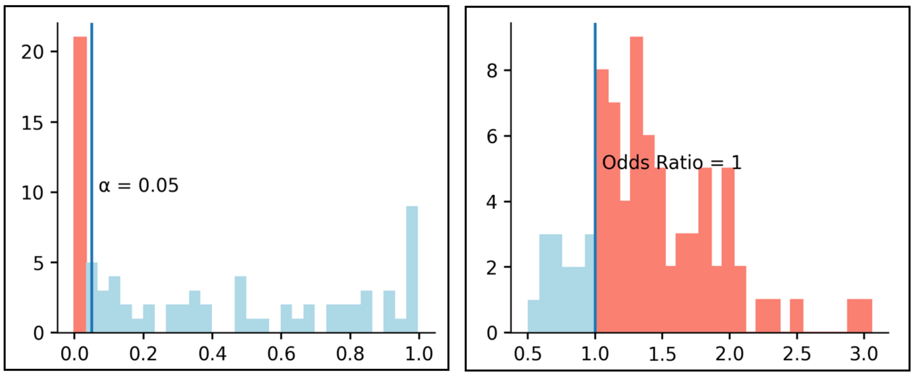


Figure 5.2: Distribution of p-values and odd ratios for the New Test Developers vs. Degrade Consistency chi-squared tests

Table 5.3: New Test Developer vs Degrade Consistency Chi-square test results (**RQ3.2**): the test-related commits (TRCs) analyzed, observed counts, p-value, and odds ratio (OR) of the projects which reject the null hypothesis after applying the Holm-Bonferroni correction.

Project (owner/repo)	**TRCs**	**New/Degrade**	**Existing/Degrade**	**p-value**	**OR**
alibaba/fastjson	2071	95 (42%)	489 (26%)	4.74e-6	2.03
apache/logging-log4j2	3416	60 (30%)	584 (18%)	1.23e-4	1.96
mockito/mockito	1995	91 (34%)	395 (23%)	2.17e-4	1.78
netty/netty	3128	229 (36%)	677 (27%)	6.63e-5	1.51
SeleniumHQ/selenium	5168	91 (23%)	619 (13%)	1.67e-7	2.01
spring-projects/spring-batch	2869	71 (39%)	652 (24%)	8.74e-5	1.96
undertow-io/undertow	1439	63 (35%)	236 (19%)	3.24e-6	2.33

5.2.1 Peripheral Developer vs. Degrade Consistency

The left plot in Figure 5.1 displays the distribution of p-values from the Peripheral Developer vs. Degrade Consistency chi-squared tests performed on the 80 test suites. Twenty-five out of the 80 suites (28%) yielded a p-value less than 0.05. However, after applying the Holm-Bonferroni correction, we only rejected the null hypothesis for eight test suites (see Table 5.2). For each of the eight test suites, the observed relative frequencies suggest that peripheral developers are more likely than core developers to commit changes that degrade consistency. For the eight projects for which we detected a significant effect, peripheral developers have between 1.20 and 2.83 greater odds of degrading the consistency of the test suite.

Despite not rejecting the null hypothesis, 50 of the remaining 72 (69%) test suites had an odds ratio greater than one (see Figure 5.2, right plot). Hence, for most test suites, peripheral developers contributed relatively more consistency-degrading commits.

5.2.2 New Test Developer vs. Degrade Consistency

The left plot in Figure 5.2 shows the distribution of p-values from the New Test Developer vs. Degrade Consistency chi-squared tests performed on the sampled test suites. The p-value was less than 0.05 for 22 (31%) test suites. After applying the Holm-Bonferroni correction, we rejected the null hypothesis for the seven test suites listed in Table 5.3.

The observed relative frequencies in each test suite imply that new test developers are more likely than existing test developers to contribute consistency-degrading commits. Depending on the projects, new test developers have between 1.51 and 2.33 greater odds of contributing degrading commits.

The odd ratios were greater than one for 59 of the remaining 73 (81%) test suites for which we failed to reject the null hypothesis (see Figure 5.2, right plot). Hence, new test developers contributed relatively more consistency-degrading commits for most test suites.

5.2.3 Example Consistency Degrading Commits

As an example, "Joe Bloggs <joebloggs@unknown>"[3] contributed a consistency-degrading commit to the apache/logging-log4j2 project in May 2013. Joe Bloggs was a peripheral and a new test developer at the time; this was his first test-related commit. The commit degraded consistency by adding more convention clashes in two test convention classes: Testing Exceptions and Assertion Density. For the Testing Exceptions class, there were three and one occurrences of the try-catch and expected attribute conventions, respectively, before the commit. Thus, the pre-commit entropy-inspired consistency for the Testing Exceptions class was 0.552. Joe Bloggs added two more occurrences of the expected attribute convention, decreasing the consistency to 0.500. For the Assertion Density class, there were 223, 132, and 114 occurrences of the one assertion per method, few assertions per method (2-3), and many assertions per method (>3) conventions, respectively. The pre-commit entropy-inspired consistency for the Assertion Density class was thus 0.397. Joe Bloggs added 4, 4, and 11 occurrences of the one, few, and many assertions per method conventions, respectively. By adding many occurrences of the least-popular alternative (i.e., many assertions per method), the commit increased uncertainty, thereby decreasing the consistency to 0.395.

As another example, "John Doe <johndoe@unknown>" contributed a commit to the apache/logging-log4j2 project that degraded consistency in August 2014. At the time, John Doe was a peripheral developer. Before the commit, 233 test class names (95%) adhered to the convention [ClassName]Test. The entropy-inspired consistency for the Naming Test

[3]The examples are from existing commits in our data set. However, we use pseudonyms and do not reveal the commit hash to protect the authors' identities.

Classes class was 0.761. This high level of consistency may be attributable to Log4j2's code style guidelines[4], which states that developers should name test classes using the [ClassName]Test convention (committed on May 19, 2014, 387ed34). Nevertheless, as a peripheral developer, John Doe added a test class called JmsAppenderIT, which does not adhere to the target convention. As a result, the entropy-inspired consistency for the Naming Test Classes class dropped to 0.752.

5.3 Threats to Validity

We are interested in assessing a developer's level of commitment to a project (construct) and are using objective commit-based contribution metrics. Our metrics may not accurately capture participation levels since they do not account for non-commit-based efforts, e.g., managing the design and evolution of the project. We opted for commit-based metrics because test conventions result from various local decisions. They are implementation-level details, especially when not formally documented. As such, we decided to focus on the developers' involvement in the code base. Commit-based metrics also make the study tractable and have been widely used in past work [57]. Moreover, we used two different metrics instead of one to mitigate the risk of incorrectly capturing a developer's participation level. This decision was important because the two metrics identified different projects.

A threat to internal validity is the incomplete author identity resolution. Developers may use various (name, email) combinations when authoring commits. Linking the various (name, email) combinations a developer uses is necessary to classify their participation level correctly. Author identity resolution, however, is a complex research problem in itself [58]. We attempted to mitigate this threat by performing email and name unification using heuristics and acknowledge that our approach may inaccurately classify some developers' participation levels. Another threat to internal validity is cross-project contributions. Developers may contribute to multiple projects within an organization (e.g., apache/deltaspike, apache/camel-quarkus). If the projects within an organization adhere to similar test conventions, our project-specific participation level operationalizations may underestimate developers' participation.

[4]https://logging.apache.org/log4j/2.x/javastyle.html

5.3 Threats to Validity

A threat to external validity stems from our choice to focus on Java JUnit test suites. We do not expect our results to generalize to other test suites and scope our claims accordingly.

6
Related Work

This section reviews past research that is most related to the research done in this thesis. Prior work has surveyed developers to identify characteristics of high-quality tests (Section 6.1) and assess the state of existing unit tests (Section 6.2). Past work has also devised metrics to evaluate test quality (Section 6.3). Throughout the subsections, we highlight how this thesis relates to and differs from the relevant prior research on software testing.

6.1 Characteristics of High-quality Tests

Multiple empirical studies surveyed developers to understand practitioners' perspectives on what constitutes high-quality unit tests. Kochhar et al. [4] conducted open-ended interviews with 21 industry and open-source practitioners, identifying 29 hypotheses that describe characteristics of good test cases. They surveyed 261 practitioners from various small to large companies and open-source projects across 27 countries to validate these hypotheses. The key finding in relation to this thesis is that most practitioners believe test code should be well-written and follow a consistent coding style. More than 96% of respondents believe that test cases should be readable and understandable, yet some voiced difficulties in keeping unit tests clean.

Similarly, Bowes et al. [3] conducted a two-day semi-structured workshop with industry partners to elicit testing principles that produce high-quality tests. Based on the workshop, the authors' experience teaching software testing courses, and content from relevant practitioner books, the authors constructed a list of 15 testing principles. The second principle in the list is "Readability and Comprehension". Grano et al. [1] also found that test readability is crucial to facilitate debugging and maintenance tasks when surveying 70 practitioners. These findings align with multiple unit testing doctrines that include readability in their guiding principles [5, 6].

Tan et al. [59] proposed a tentative list of 15 criteria for "good" test cases, along with a ranking of their relative importance. They constructed the list of criteria by combining past research and insights from employees at three partner companies. The criteria were then evaluated and ranked by 13 Swedish experts in the software testing industry. The results revealed that developers deem "Maintainable" and "Consistent" to be significant criteria for good tests, with average rankings of 7.4 and 6.8 out of 10, respectively, where ten indicates "extremely relevant".

These studies motivate this thesis by highlighting that practitioners perceive readability and consistency as essential signals for test quality. In terms of methodology, this thesis is similar in that it emphasizes the importance of the practitioner's perspective. Rather than surveying developers to identify testing principles, we survey relevant grey literature to identify test conventions.

6.2 State of Existing Unit Tests

Despite practitioners' agreement that tests should be readable, various empirical studies found that existing tests are far from it. Grano et al. [7] performed an exploratory study comparing the readability of manually written tests to the classes they test for three popular Apache projects. They used a state-of-the-art readability model to compute the readability of the tests and production classes. They found that source code is significantly more readable than test code, suggesting developers tend to neglect the readability of tests.

As such, it is no surprise that Li et al. [8] found that more than half of the 212 developers surveyed experience "moderate" to "very hard" difficulty understanding unit tests. Likewise, by surveying 225 developers, Daka and Fraser [9] identified difficulty understanding tests as a top obstacle to fixing failing tests.

These studies suggest that developers neglect or struggle to write readable unit tests. They highlight the need for more research aiming to improve test quality. With this thesis, we strive to do just that by exploring test convention consistency as a proxy for readability. We also design and develop Teslo to provide tool support to developers seeking to improve the quality of their test suites.

6.3 Test Quality Metrics

Past work has proposed different metrics to measure test quality. Most of the proposed metrics focus on assessing test code's ability to perform one of its three purposes: detect faults. Of all such metrics, code coverage [60] was one of the first invented and is the most widely researched and embraced by practitioners. Code coverage—the ratio of production code executed by test code—is easy to compute and interpret. Its main limitation is that it verifies that different code paths are executed, not that the correct behavior is tested. Another well-researched metric is mutation score [61]. Mutation score measures the percent of artificially injected defects a test suite can detect. Although mutation score is more effective than code coverage [62], it is computationally expensive and, therefore, rarely adopted in the industry.

In addition to their limitations, these two metrics suffer from a limited scope. Specifically, they fail to capture test code's ability to perform two of its three intended purposes: act as documentation and drive debugging. For this reason, Grano et al. [1] discovered that practitioners consider code coverage insufficient as a test quality metric.

One work that attempts to address this gap is that of Daka et al. [63]. They designed a domain-specific model to measure unit test readability. The model is based on human judgements and uses various features such as unused identifiers, assertions, and method diversity. They use the model to generate test suites with improved readability automatically. Human annotators preferred their enhanced tests to those generated by EvoSuite and could answer maintenance questions quicker with the same accuracy.

Another line of research that addresses this gap is that of test smells. In 2001 Deursen et al. [64] proposed a set of 11 test code smells along with refactorings to mitigate them. For example, the test smell "Assertion Roulette" is when a test method has many assertions without explanation, making it challenging to debug failing tests. The suggested refactoring uses JUnit's optional String argument to provide an explanatory message to the user when the assertion fails.

Since their proposition, different works have sought to build tools to detect test smells [65, 46, 66, 67] and investigate their relationship with various factors, such as error-proneness of source code [45]. One particular work by Bavota et al. [68] investigated whether the presence of test smells impacts program comprehension during maintenance tasks. They asked participants with varying levels of experience—from bachelor's students to industry professionals—to perform program comprehension tasks on tests with and without test smells. They found that test smells have a strong negative impact on program comprehension and maintenance. Hence, the number of test smells in a test suite could be used as a signal for its readability.

This thesis complements the above works by attempting to design a metric that better captures the readability of test code. Domain-specific test readability metrics, and other ensemble metrics, can leverage our proposed test convention consistency metrics as a feature within their models. Moreover, the problem of test convention inconsistency is in itself a sort of test-suite-level smell.

7
Conclusion

We introduced and explored test convention consistency as one dimension of test quality. To that end, we performed a three-staged investigation. First, we conducted a grey literature survey to elicit a catalogue of test convention classes discussed by the Java JUnit developer community. As a result, we identified 61 unique JUnit test conventions, which we organized into a catalogue of 23 test convention classes. The catalogue captures otherwise tacit knowledge that can facilitate the design and development of test generation and refactoring tools and ASATs. There may be a discrepancy between the test conventions developers promote and use in practice. Hence, we can improve the catalogue with a second study that examines open-source test suites to identify test conventions.

By analyzing the content of the online blog posts and tutorials, we also found that Java JUnit developers consider test method name conventions important to unit test readability. Many of them stressed the importance of consistency, validating test convention consistency as a dimension of test quality. The results also highlighted the need for tools to measure consistency and detect inconsistency by revealing that convention classes are likely in team projects.

Next, using the resulting catalogue, we designed and developed Teslo—an open-source prototype tool to measure and visualize the consistency of Java JUnit test suites. In doing so, we posit two novel metrics to measure test convention consistency—one accuracy-based, the other entropy-inspired—each effective for different use cases. Neither metric requires developers to have a priori knowledge of their team's preferred conventions, facilitating adopting. Future research can leverage these metrics to enhance readability and other test quality metrics.

Teslo presents the consistency scores in a hierarchical user interface that facilitates comprehension and action-taking. Depending on their informational needs, developers can drill down to view the consistency at the package, class, or test level. Although the current version of the tool focuses on Java and JUnit test conventions, the tool's architecture and guiding principles are transferable to any type of test convention.

Lastly, we sought to study factors that impact the evolution of test convention consistency. As a first step, we performed a multi-case study to determine whether a relationship exists between developer participation level and the evolution of test convention consistency. We found that developers with lower participation levels are more likely to degrade test suite consistency in 14% of the analyzed projects. In those projects, developers with lower participation levels had between 1.20 and 2.83 greater odds of contributing consistency-degrading commits. In over 90% of the test suites for which we found no significant effect, developers with low participation levels still contributed relatively more consistency-degrading commits.

Bibliography

[1] G. Grano, C. De Iaco, F. Palomba, and H. C. Gall, “Pizza versus pinsa: On the perception and measurability of unit test code quality,” in *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution*, 2020, pp. 336–347.

[2] G. Meszaros, *Xunit test patterns: Refactoring test code*. Addison-Wesley, 2007.

[3] D. Bowes, T. Hall, J. Petric, T. Shippey, and B. Turhan, “How good are my tests?” in *Proceedings of the 8th IEEE/ACM Workshop on Emerging Trends in Software Metrics*, 2017, pp. 9–14.

[4] P. S. Kochhar, X. Xia, and D. Lo, “Practitioners’ views on good software testing practices,” in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, 2019, pp. 61–70.

[5] R. C. Martin, *Clean code A handbook of agile software craftmanship*. Prentice Hall, 2009.

[6] L. Koskela, *Effective unit testing: A guide for Java developers*. Manning Publications, 2013.

[7] 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 26th IEEE Conference on Program Comprehension*, 2018, p. 348–351. [Online]. Available: https://doi-org.proxy3.library.mcgill.ca/10.1145/3196321.3196363

[8] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, “Automatically documenting unit test cases,” in *2016 IEEE International Conference on Software Testing, Verification and Validation*, 2016, pp. 341–352.

[9] E. Daka and G. Fraser, “A survey on unit testing practices and problems,” in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering*, 2014, pp. 201–211.

[10] “Code conventions for the java programming language,” Apr 1999. [Online]. Available: https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

[11] M. Nordberg, “Managing code ownership,” *IEEE Software*, vol. 20, no. 2, pp. 26–33, 2003.

[12] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, 2013, p. 672–681.

[13] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, “Analyzing the state of static analysis: A large-scale evaluation in open source software,” in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, 2016, pp. 470–481.

[14] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, “Context is king: The developer perspective on the usage of static analysis tools,” in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 38–49.

[15] M. P. Robillard, *Introduction to Software Design with Java*. Springer Nature, 2022.

[16] A. Zerouali and T. Mens, “Analyzing the evolution of testing library usage in open source java projects,” in *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 417–421.

[17] S. Gulati and R. Sharma, *Chapter 2. Understanding Core JUnit 5*. Apress, 2017.

[18] V. Garousi, M. Felderer, and M. V. Mäntylä, “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering,” *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[19] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, *Benefitting from the Grey Literature in Software Engineering Research.* Springer International Publishing, 2020, pp. 385–413.

[20] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? a multi-vocal literature review," *Information and Software Technology*, vol. 76, pp. 92–117, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916300702

[21] S. Overflow, "What types of questions should i avoid asking?" 2022. [Online]. Available: https://stackoverflow.com/help/dont-ask

[22] E. Pariser, *The filter bubble: what the Internet is hiding from you.* Viking, 2012.

[23] A. Odlyzko, "Network neutrality, search neutrality, and the never-ending conflict between efficiency and fairness in markets," *Review of Network Economics*, vol. 8, no. 1, p. 40–60, Mar 2009.

[24] S. Trieflinger, J. Münch, E. Bogazköy, P. Eißler, J. Schneider, and B. Roling, "How to prioritize your product roadmap when everything feels important: A grey literature review," in *IEEE International Conference on Engineering, Technology and Innovation*, 2021, pp. 1–9.

[25] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301527

[26] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217303060

[27] A. Kumar, "7 popular strategies: Unit test naming conventions," Jun 2021. [Online]. Available: https://dzone.com/articles/7-popular-unit-test-naming

[28] O. Stefanovskyi, "Unit test naming conventions," Jan 2019. [Online]. Available: https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea

[29] J. Reid, "Unit test naming: The 3 most important parts," Jan 2022. [Online]. Available: https://qualitycoding.org/unit-test-naming/

[30] P. Kainulainen, "Writing clean tests - naming matters," Jan 2018. [Online]. Available: https://www.petrikainulainen.net/programming/testing/writing-clean-tests-naming-matters/

[31] J. Reese, "Best practices for writing unit tests - .net," Nov 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

[32] V. Khorikov, "You are naming your tests wrong!" Aug 2019. [Online]. Available: https://enterprisecraftsmanship.com/posts/you-naming-tests-wrong/

[33] C. Vasquez, "Introduction to unit testing with java," Jun 2018. [Online]. Available: https://dev.to/chrisvasqm/introduction-to-unit-testing-with-java-2544

[34] S. A. Smith, "Unit test naming convention," Jan 2011. [Online]. Available: https://ardalis.com/unit-test-naming-convention/

[35] R. Fadatare, "Junit framework best practices," Aug 2018. [Online]. Available: https://www.javaguides.net/2018/08/junit-framework-best-practices.html

[36] N. H. Minh, "Junit tutorial for beginner with eclipse," Aug 2019. [Online]. Available: https://www.codejava.net/testing/junit-tutorial-for-beginner-with-eclipse

[37] V. Farcic, "Test driven development (tdd): Best practices using java examples," Dec 2013. [Online]. Available: https://technologyconversations.com/2013/12/24/test-driven-development-tdd-best-practices-using-java-examples-2/

[38] L. Gupta, "Junit best practices guide," Dec 2020. [Online]. Available: https://howtodoinjava.com/best-practices/unit-testing-best-practices-junit-reference-guide/

[39] P. Hauer, "Modern best practices for testing in java," Jun 2022. [Online]. Available: https://phauer.com/2019/modern-best-practices-testing-java/

[40] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," 2014, p. 281–293. [Online]. Available: https://doi-org.proxy3.library.mcgill.ca/10.1145/2635868.2635883

[41] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *ommunications of the ACM*, vol. 61, no. 4, p. 58–66, mar 2018. [Online]. Available: https://doi-org.proxy3.library.mcgill.ca/10.1145/3188720

[42] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension*, 2019, pp. 209–219.

[43] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2021.

[44] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[45] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, 2018, pp. 1–12.

[46] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 1650–1654. [Online]. Available: https://doi.org/10.1145/3368089.3417921

[47] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production amp; test code," in *Proceedings of the 1st IEEE International Conference on Software Testing, Verification, and Validation*, 2008, pp. 220–229.

[48] JavaParser, "Javaparser," 2008. [Online]. Available: https://javaparser.org/

[49] A. Zerouali and T. Mens, "Analyzing the evolution of testing library usage in open source java projects," in *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 417–421.

[50] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, p. 92–101. [Online]. Available: https://doi-org.proxy3.library.mcgill.ca/10.1145/2597073.2597074

[51] O. Jarczyk, S. Jaroszewicz, A. Wierzbicki, K. Pawlak, and M. Jankowski-Lorek, "Surgical teams on github: Modeling performance of github project development processes," *Information and Software Technology*, vol. 100, pp. 32–46, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491730304X

[52] K. Crowston, K. Wei, Q. Li, and J. Howison, "Core and periphery in free/libre and open source software team communications," in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, vol. 6, 2006, pp. 118a–118a.

[53] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, p. 309–346, jul 2002. [Online]. Available: https://doi-org.proxy3.library.mcgill.ca/10.1145/567793.567795

[54] T. Dinh-Trong and J. Bieman, "Open source software development: a case study of freebsd," in *Proceedings of the 10th International Symposium on Software Metrics*, 2004, pp. 96–105.

[55] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, pp. 2035–2071, 2015.

[56] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70, 1979.

[57] M. Joblin, S. Apel, C. Hunsen, and W. Mauerer, “Classifying developers into core and peripheral: An empirical study on count and network metrics,” in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*, 2017, pp. 164–174.

[58] T. Fry, T. Dey, A. Karnauch, and A. Mockus, “A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, p. 518–522. [Online]. Available: https://doi.org/10.1145/3379597.3387500

[59] H. Tan, V. Tarasov, and A. Adlemo, “Test case quality as perceived in sweden,” in *Proceedings of the 5th IEEE/ACM International Workshop on Requirements Engineering and Testing*, 2018, pp. 9–12.

[60] J. C. Miller and C. J. Maloney, “Systematic mistake analysis of digital computer programs,” *Communications of the ACM*, vol. 6, no. 2, p. 58–63, feb 1963. [Online]. Available: https://doi-org.proxy3.library.mcgill.ca/10.1145/366246.366248

[61] R. DeMillo, R. Lipton, and F. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[62] N. Li, U. Praphamontripong, and J. Offutt, “An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage,” in *Proceedings of the 2nd IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 220–229.

[63] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, “Modeling readability to improve unit tests,” in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, p. 107–118. [Online]. Available: https://doi.org/10.1145/2786805.2786838

[64] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok, “Refactoring test code,” *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering*, p. 92–95, 2001.

[65] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*, 2013, pp. 322–331.

[66] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[67] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, 2018, pp. 311–322.

[68] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, p. 1052–1094, aug 2015. [Online]. Available: https://doi-org.proxy3.library.mcgill.ca/10.1007/s10664-014-9313-0