Understanding Test Convention Consistency as a Dimension of Test Quality

MARTIN P. ROBILLARD, MATHIEU NASSIF, and MUHAMMAD SOHAIL, McGill University, Canada

Unit tests must be readable to help developers understand and evolve 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. We investigated test convention consistency as a dimension of test quality. We formalized test suite consistency as the extent to which alternatives are used within a code base and introduce two complementary metrics to capture this extent. We elaborated a catalog of over 30 test conventions for the Java language organized in 10 convention classes that group mutual alternatives. We developed tool support to detect occurrences of conventions, compute consistency metrics over a test suite, and view occurrences of conventions in the corresponding code. We applied our tools to study the consistency of the test suites of 20 large open-source Java projects. The study validates the design of the test convention classes, provides descriptive statistics on the range of consistency values for ten different convention classes, and enables us to link observed changes in consistency values to specific events in the change history of our target systems, thus providing evidence of the construct validity of the metrics. We conclude that analyzing test suite consistency via static analysis shows promise as a practical approach to help improve test suite quality.

CCS Concepts: • **Software and its engineering → Automated static analysis**; **Consistency**; *Software maintenance tools*; *Empirical software validation.*

Additional Key Words and Phrases: Software Testing, Test Quality, Test Conventions, Inconsistency Detection.

ACM Reference Format:
Martin P. Robillard, Mathieu Nassif, and Muhammad Sohail. 2023. Understanding Test Convention Consistency as a Dimension of Test Quality. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2023), 38 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Unit tests serve many purposes: they help detect faults, act as documentation, and facilitate debugging activities [16, 36]. The multi-purpose nature of unit tests makes it difficult to define what constitutes a high-quality test. Intuitively, a unit test should measurably fulfill each of its purposes. Devising a metric that captures a test's effectiveness at achieving all three of its purposes is challenging because the factors influencing each purpose do not necessarily align. For example, using a descriptive name makes a test more effective as documentation and facilitates debugging, but does not affect the test's ability to detect faults.

Researchers have proposed various metrics to estimate test quality. Most of these metrics evaluate the tests' 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 [16]. They believe code coverage paints an incomplete picture of

Authors' address: Martin P. Robillard, robillard@acm.org; Mathieu Nassif, mathieu.nassif@mail.mcgill.ca; Muhammad Sohail, muhammad.sohail@mail.mcgill.ca, School of Computer Science, McGill University, Montréal, QC, Canada.

test quality as it fails to assess the 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 a more elusive problem. 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 tests to support documentation and maintenance, developers must be able to effectively read and understand them. Past studies confirm this intuition by showing that practitioners deem readability crucial to achieving high-quality tests [6, 16, 27]. This finding also aligns with various unit testing doctrines that include readability in their guiding principles [28, 35].

Despite the perceived importance of readability, test code is significantly less readable than production code in practice [17]. For example, Li et al. [30] 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 [8] identified difficulty understanding tests as a main 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" [40]. It is not necessarily the chosen conventions themselves that improve readability but rather the uniformity they yield that reduces the cognitive load required to understand the code (see Section 6.1).

We investigate *test convention consistency* as one dimension of test quality. Any test suite may follow *test conventions* that are alternative of each other. For example, one convention for naming test classes is to prefix the class name with the string `"Test"`, whereas an alternative convention is to use `"Test"` as a suffix instead. Alternative conventions naturally form *test convention classes* (i.e., all alternative conventions for naming test classes). We formalize test suite consistency as the extent to which the same alternative is used within a test convention class and introduce two complementary metrics we designed to capture this extent. One metric, termed *accuracy-based*, relies on identifying a preferred, or dominant, convention as a target. A second metric, termed *entropy-based*, leverages the concept of entropy to measure the amount of uncertainty about which alternative of a convention class is used.

We designed a catalog of convention classes for Java projects. These convention classes cover different aspects of the design and implementation of unit testing, including naming test classes and methods, naming variables that store oracles and results in test implementations, test and assertion documentation, different aspects of testing for exceptional behavior, and more. Each convention class in our catalog lists a number of alternative conventions, in most cases mentioned in the grey literature on unit testing. We designed each convention class to provide an exhaustive set of mutually-exclusive alternative conventions. For each convention class, we included background that explains why we selected this convention and how it is relevant, design information that clarifies what the convention alternatives map to in practice, and additional discussion and insights on the design choices we made.

Numerous important design decisions come into play when attempting to detect and interpret occurrences of test conventions. We investigated the feasibility of supporting test convention consistency in practice by developing a suite of tools that consists of a static analyzer for computing consistency data (TestComet) and a web application for allowing users to obtain and view this data (Teslo). Developers can leverage a tool such as 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. Moreover, we designed the tools to facilitate customizations and understandability, mitigating two commonly cited barriers to adopting automated static analysis tools [3, 23, 63].

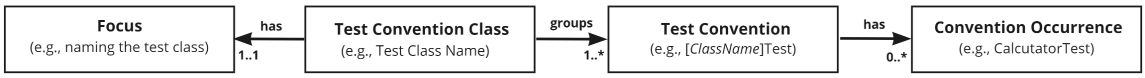


Fig. 1. Test convention consistency concepts and their relationships.

Finally, we demonstrate how our consistency metrics enable methodical reasoning about the evolution of test suites. Specifically, we report on a multi-case study of JUnit test suites for 20 notable open-source Java projects to explore the potential of consistency analysis for quality assessment of evolving test suites. As part of this exploration, we performed a detailed comparative analysis of the relation between the two metrics, analyzed global trends in the evolution of test suite consistency, and analyzed specific consistency change events in the evolution of the projects. We report on the consistency values we observed in practice for our catalog of convention classes, as well as more specific insights about the relation between observed changes in metric values and specific events in the development history of the systems under study. Ultimately, the case studies show how consistency metrics allow to reason more precisely and quantitatively about one important aspect of test quality, thereby promoting awareness of its importance and providing technical means for improving it.

This article makes four main contributions, each presented in a separate section. Section 2 provides the details of our first contribution, namely a conceptual framework and corresponding metrics for measuring test convention consistency. Section 3 provides our catalog of test convention classes. Section 4 contributes the design of TestComet and Teslo, the tool infrastructure we developed to explore test convention consistency in practice. In Section 5, we report on the empirical study of the consistency of 20 large Java projects. Finally, Section 6 discusses related work and Section 7 concludes.

2 DEFINING TEST CONVENTION CONSISTENCY

We define the testing and related concepts we rely on and introduce two metrics to capture test convention consistency in practice.

2.1 Problem Formulation

We cast the problem of measuring test convention consistency in the context of a software project that includes a collection of *unit tests* we refer to as a *test suite*. The test suite may follow *test conventions*, some of which may be *alternatives* of each other.

We define a *test convention* as a rule for implementing a specific aspect of a unit test or related code element. We limit the scope of our work to conventions that can be *automatically* and *unambiguously* detected. For example, a rule prescribing that the name of a class declaring unit tests should be composed of the name of the class being tested followed by the word Test is an applicable test convention. In contrast, the principle that unit tests should execute fast [35] does not qualify as a convention applicable to consistency checking. We distinguish between a *test convention* (the rule) and a test convention *occurrence* (an instance of application of the rule). For example, a test class named CalculatorTest is an occurrence of the test convention described above, which we refer to as *Test Suffix*.

Conventions can be *alternatives* of each other. We consider two or more test conventions to be alternatives if they have the same *focus*, namely to achieve a specific code organization goal usually targeting the same code element (e.g., two conventions to name the class declaring the unit tests). We group alternative conventions into *test convention*

classes. An example of test convention class is *Test Class Name*, which groups alternatives for naming the test class. As we have seen, one possible convention is *Test Suffix*, and another convention is *Test Prefix*. Figure 1 summarizes these concepts and how they relate to each other.

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

2.2 Consistency Metrics

We introduce two complementary metrics to measure test convention consistency. Our primary goal was to design test suite consistency metrics that do not require developers to have a priori knowledge of their team's preferred conventions.

Accuracy-based Test Suite Consistency (Consistency$_A$). To define an accuracy-based metric, we must identify one *target convention* per convention class. We define the target convention to be the most frequent. The occurrence of any other convention within the convention class constitutes an inconsistency. For example, if a test convention class TCC has three conventions tc_1, tc_2, tc_3 with, respectively, 50, 25, and 25 occurrences, then tc_1 is the target convention.[1] Given fdist(TCC) as the distribution of occurrence frequencies for a convention class TCC, we define our accuracy-based metric as:

$$\text{Consistency}_A(TCC) = \frac{\max(\text{fdist}(TCC))}{\text{sum}(\text{fdist}(TCC))}$$

In our example, we thus have $\text{Consistency}_A(TCC) = 50/(50+25+25) = 0.5$. The accuracy-based metric is prescriptive and useful in cases where a team wants to enforce a single convention per convention class. However, the metric is only sensitive to the number of occurrences of the target convention. As such, it is not sensitive to different occurrence distributions. For example, the accuracy-based metric fails to differentiate between the following two test convention classes, each of which has four conventions but different occurrence distributions: $TCC_1 = \{50, 20, 15, 15\}$ and $TCC_2 = \{50, 49, 1, 0\}$. In both cases, the metric yields a value of 0.5. Yet, the inconsistency exhibited by TCC_2 is arguably less detrimental to readability. Our other metric mitigates this shortcoming.

Entropy-based Test Suite Consistency (Consistency$_E$). The second metric leverages the concept of entropy. Shannon's entropy [51] is a nonnegative value that captures the amount of informational value (or "surprise", or "uncertainty") communicated by a random variable. In our context, increasing the amount of information is undesirable as it requires developers to spend more effort when reading the test suite. Because the additional informational value does not affect the purpose or execution of the test, it constitutes noise. Thus, it is desirable to minimize entropy.

We illustrate this relation intuitively using a hypothetical test convention class TCC that comprises four conventions. TCC has maximal entropy (and thus minimum consistency) when all four conventions occur with equal probability (i.e., 25% of the time). In this case, developers can be maximally uncertain of which convention to expect when reading an arbitrary test.

[1]The calculation of the metric is not affected by cases where two or more conventions share a maximum frequency value, since the value of the metric is only impacted by the maximum frequency value, independently of which convention class exhibits this frequency. In any case, a team wishing to improve the consistency of their tests suite will always have to decide which convention to enforce among alternatives, whether they are equally frequent or not.

Conversely, *TCC* has minimum entropy (and thus maximum consistency) when one of the four conventions occurs 100% of the time. The developer is never surprised as they always encounter the same convention. Hence, the lower the entropy of a test convention class, the more consistent it is.

In the context of convention classes, Shannon's entropy is defined as

$$\text{Entropy}(TCC) = -\sum_{tc} p(tc) log(p(tc))$$

where tc denotes a test convention in TCC, $p(tc)$ is the probability of observing tc, and log is the base 2 logarithm.[2] In other words, the entropy is the arithmetic mean (i.e., average) of $-log(p(tc))$, weighted by the probabilities themselves.

One way to interpret entropy is to consider its exponential. A test convention class with an entropy E is as inconsistent as a hypothetical class for which developers alternate randomly between 2^E conventions with uniform probability. Thus, an entropy of 1 is equivalent to developers using $2^1 = 2$ conventions, each for half of the tests, whereas an entropy of 3 is as inconsistent as developers alternating between $2^3 = 8$ conventions.[3]

We derive a metric of consistency by taking the inverse of the previous exponential:

$$\text{Consistency}_E(TCC) = \frac{1}{2^{\text{Entropy}(TCC)}} = 2^{-\text{Entropy}(TCC)} = \prod_{tc} p(tc)^{p(tc)}$$

where $\prod$ denotes the product operator. Similarly to Consistency_A, this metric takes values between 0 (exclusive) and 1 (inclusive), with higher values denoting more consistent test suites.

The product form in the formula above shows another interpretation of this metric: it is the geometric mean of the probabilities, weighted by the probabilities. Thus, it represents the entire distribution of a test convention class, with an emphasis on more frequent conventions. It follows from this observation that Consistency_E cannot be higher than Consistency_A. Both consistency metrics return the same value only in situations where N conventions are each used as often as each other (the value in such situation is $1/N$). We provide a detailed examination of the relation between the two metrics in Section 5.3.

If we revisit the examples above, we get $\text{Consistency}_E(TCC_1 = \{50, 20, 15, 15\}) = 0.30$ and $\text{Consistency}_E(TCC_2 = \{50, 49, 1, 0\}) = 0.48$, which follows our intuition that TCC_2 is more consistent than TCC_1.

2.3 Target Elements for Convention Classes

A crucial concept for the calculation of test suite consistency is that of a *convention target element*. For uniformity, each convention class must be designed to apply to an explicitly-defined collection of code elements. For example, the target element for the convention class Naming Test Classes is the set of all classes that contain at least one unit test. The frequency distribution fdist(TCC) for a convention class TCC should be expressed in terms of target elements and include all applicable elements. While the target element for Naming Test Classes is obvious, other test convention classes can require a more careful determination. For example, we can consider a convention to name unit tests. A common convention is to incorporate the name of the method under test (or *focal* method) in the name of the test, e.g., `testFocalMethod()`. However, how should unit tests that do not focus on a particular production method be treated? Are they applicable targets that somehow violate the convention, or are they not applicable units for the convention? The definition of convention target is thus tied to the definition of the corresponding convention class and must therefore be explicitly specified on a per-convention-class basis.

[2]Following common practice, in cases where $p(tc) = 0$ we consider that the entire term $p(tc)log(p(tc)) = 0$.
[3]These comparisons are only useful for reference. Different non-uniform distributions can have the same entropy.

2.4 Limitations

In designing metrics to capture the consistency of test suites, we face the fundamental problem that conventions are, by definition, arbitrary. For any convention class, different ways to interpret a convention can lead to different partitions between alternatives. For example, there exist two common practices for handling unexpected exceptions that occur during the execution of a unit under test: the unit test can either catch the exception and explicitly fail the test, or declare the exception in the test method's declaration and propagate the exception to the test runner, which will fail the test. However, some complex unit tests may exhibit characteristics of both. How should such a test be classified? A practical option is to incorporate a catch-all *Other* alternative in the test convention class. However, interpreting the meaning of the *Other* alternative is ambiguous. A unit test could fall into the *Other* category because of a design or implementation error in the test, because the test implements a specialized convention of the project, or because the code of the test is too complex to be reliably interpreted via static analysis. For more open-ended conventions, the problem of test convention class partition is even greater. As an example, we take the convention for naming test classes. A reasonable set of alternatives could include tc_1: using Test as a prefix (e.g., TestCalculator), tc_2: using Test as a suffix (e.g., CalculatorTest), and tc_3: *Other*. However, what about occurrences such as CalculatorTests ($tc_{3.1}$), or CalculatorTestCases ($tc_{3.2}$), or even TestingCalculator ($tc_{3.3}$). Which variants to consider as distinct alternatives has an impact on the value of the metrics.

For Consistency$_A$, there is only an impact if the *Other* category is the most frequent. For fdist$(TCC) = \{\text{tc}_1 = 50, \text{tc}_2 = 25, \text{tc}_3 = 25\}$, Consistency$_A = 0.5$ independently of whether *Other* conceptually groups multiple partitions. However, for another test suite with fdist$(TCC) = \{\text{tc}_1 = 25, \text{tc}_2 = 25, \text{tc}_3 = 50\}$, the value of Consistency$_A$ would drop from 0.5 to 0.25 if *Other* is reorganized as two sub-partitions which happen to have the same frequency. As Consistency$_E$ is influenced by the frequency of all conventions in a class, reorganizing the conventions in that class will always affect the consistency value. However, the change will only be meaningful if a frequent convention is modified.

3 A CATALOG OF TEST CONVENTION CLASSES

Experimenting with the concept of test suite consistency requires a precise definition of test convention classes. We contribute an initial catalog of ten test convention class definitions applicable to Java projects. To be of practical use in generating consistency reports, a convention class must be robust. In particular, the conventions within a class should be *exhaustive* and *mutually exclusive*. That is, all target elements of a convention class must be occurrences of exactly one convention within the class.

We define the test convention classes based on insights from the research and grey literature on test conventions. We introduce each convention class in its own section (see Table 1). Each section provides the background and motivation for the convention class, a technical description of the conventions in the class, and a discussion of the rationale for its design.

This catalog provides an initial framework for exploring test suite consistency based on a reproducible set of commonly recognized conventions. It is not possible for such a catalog to be final, universal, or complete: test conventions are applied differently in different contexts, and they keep evolving. However, this initial catalog is intended to serve as a fixed baseline for research, and can be easily adapted and extended for application in different contexts.

3.1 Naming Test Methods

This test convention class groups alternatives for naming the method that implements a unit test.

Table 1. Overview of Our Conventions Catalog

Convention Class	Details	Focus
Naming Test Methods	Sec. 3.1	Structure of the test method's name
Naming Test Classes	Sec. 3.2	Structure of the name of test classes (i.e., containers)
Naming Oracles	Sec. 3.3	Keywords in variables' names for the expected value for assertions
Naming Results	Sec. 3.4	Keywords in variables' names for the computed value for assertions
Specifying Oracles	Sec. 3.5	Ways to provide the expected value of an assertion
Supplying Failure Messages	Sec. 3.6	Use of the optional *message* argument in assertion methods
Assertion Density	Sec. 3.7	Number of assertions per test method
Testing Exceptions	Sec. 3.8	Ways to test that an expected exception is actually thrown
Handling Exceptions	Sec. 3.9	Ways to handle unexpected exceptions thrown in test methods
Documenting Test Methods	Sec. 3.10	Use of Javadoc comments to document test methods

Background. Numerous experts and commentators have proposed or illustrated various conventions for naming test methods [11, 12, 19, 24, 26, 29, 38, 45, 46, 53, 56, 59, 60, 62]. These conventions encode information about the test that varies from elementary to elaborate. Possibly the most elementary convention is to prefix the name of the method under test with the keyword test, e.g., testAdd() [38]. In contrast, an example of elaborate convention based on behavior-driven development (BDD) is given[Input]_when[Method]_then[ExpectedOutput], which encodes both the test input and the expected result of the test in addition to the method under test [29]. Conventions also vary in the exactness and checkability of the information that is encoded in the test name. For example, it is relatively straightforward and unambiguous to verify that a production method referenced in a unit test both exists and is called within the code of the unit test. Other conventions, such as test[Feature being tested] [29], likely rely on subjective abstractions (the feature) relevant to the author of the test but without a corresponding code element. Finally, there even exist conventions about how to phrase descriptions of features under test in terms of grammatical patterns [44].

Design. The target element for this convention class is any method in the test suite annotated as a test using the annotations: @Test, @ParameterizedTest and @RepeatedTest. Because it is not possible to reliably detect project feature names and similar concepts without expert project knowledge, this convention class only supports naming conventions for tests that explicitly refer to the focal method. The placeholder [MethodName] refers to a case-insensitive match between the name of the test and any method called within the test. We use a case-insensitive match despite the fact that Java names are case-sensitive because of the common practice of altering the case of focal methods for consistency with the test name (e.g., a test for method add named testAdd). The placeholder [AdditionalInfo] refers to any sequences of characters legal in a Java method name.

Convention	Description
Test Method Name	The name of the test method is test[MethodName]
Test Method Name Underscore	The name of the test method is test_[MethodName]
Test Method Name Extra	The name of the test method is test[MethodName][AdditionalInfo]
Test Method Name Extra Underscore	The name of the test method is test_[MethodName]_[AdditionalInfo]
Test Method Name Mix	The name of the test method is test[MethodName]_[AdditionalInfo]
Method Name Extra	The name of the test method is [MethodName][AdditionalInfo]
Method Name Extra Underscore	The name of the test method is [MethodName]_[AdditionalInfo]
Other Method Name	The name of the test method contains [MethodName], but does not follow any previously identified pattern.
No Focal Method	There is no case-insensitive match between the names of methods called in the test body and the test name.

Discussion. The design of this convention class hinges on the name of the focal method, but also makes provisions for the integration of additional information in a structured way. This design decision allows us to eliminate any subjectivity in the application of the convention, which is a necessary condition for automated checking. However, aggregating all additional information under one banner means we lose some fidelity in capturing fine-grained information. For example, the BDD convention mentioned above (given_...), and all similar variants, will be classified as Other Method Name. While not technically incorrect, this outcome is of limited practical usefulness. This convention class will therefore be mostly relevant to projects that employ basic conventions centered around traceability to the focal method.

3.2 Naming Test Classes

This test convention class groups practices for naming classes that define unit tests.

Background. In JUnit, test classes explicitly represent a unit of organization for unit tests. Test classes do not typically group arbitrary tests, but instead map to specified aspects of the code's organization. A common practice is to define a directory structure for tests that parallels the structure of production code [11, 12]. Test classes typically group either tests "which tests the methods of a single class", or "which ensure that a single feature is working properly" [24]. In both cases, a naming convention for the test class helps emphasize what the tests located in the test class have in common.

Design. The target element for this convention class is any class in the test suite (defined as a class containing at least one unit test). This convention class addresses exclusively the lexical aspect of the naming of test classes, and in particular the placement and morphology of the keyword Test within the name.

Convention	Description
Test Prefix	The name of the class starts with Test
Test Postfix	The name of the class ends with the singular Test
Tests Postfix	The name of the class ends with the plural Tests
Contains Test	The name of the class contains the string Test, but not at the beginning or end
No Test	The name of the class does not contain the string Test

Discussion. The design of this convention class does not incorporate any evaluation of the target of the class because of the ambiguity involved. For example, a class named CalculatorTest [38] could be intended to contain tests for the methods of a class Calculator, a *calculator* feature, or the combination of both (i.e., a *calculator* feature implemented via a Calculator class and additional classes). Including a validation of the unit under test as part of this convention would, in the limited cases where it is possible, also complicate the interpretation of violations by mixing rules about test names with rules about test design. The purpose of this convention class is thus exclusively to identify inconsistencies in the use of the keyword Test in the class's name.

3.3 Naming Oracles

This test convention class groups practices for naming the variable storing the oracle, i.e., the expected value of an assertion.

Background. Some experts advocate to store the expected outcome of a test in a local variable whose name involves the keyword expected. Two prominent alternatives include prefixing the variable name with expected [20, 53] or using the keyword without additional text [62].

Design. The target element for this convention class is any *assertion* that involves an oracle–result pair whose oracle is stored in a variable. We define an assertion to be any call to a JUnit *assertion method*.[4]

Convention	Description
Expected	The name of the oracle is a case-insensitive match for expected
Expected Prefix	A proper prefix of the name of the oracle is a case-insensitive match for expected
Other	All other naming schemes for the oracle variable

Discussion. Our sample implementation is limited to JUnit assertions. Support for other testing frameworks is outside the scope of this project. It is also not possible to anticipate all possible user-defined assertion methods, but for the practical deployment of the approach it would be straightforward to add matching rules to include project-specific assertions. Our implementation also considers only the two naming conventions we could trace to public recommendations. The implication is that any convention that does not involve the keyword expected will effectively not be checked, and for applicable projects the consistency of this convention class is likely to be close to 1.0 (100%). For experimental purposes this outcome can be manually flagged as an inapplicable convention. For field use, it would be necessary to encode the specialized convention used by the project for this convention class to be useful.

[4]Examples of assertion methods that involve an oracle–result pair are assertEquals and assertSame. Examples of assertion methods that do not involve an oracle–result pair are assertTrue and assertFalse.

3.4 Naming Results

This test convention class groups practices for naming the variable storing the result of executing the unit under test. Its intent and design parallels that of the *Naming Oracles* convention class.

Background. Some experts advocate to store the actual outcome of a test in a local variable whose name involves the keyword actual. Two prominent alternatives include prefixing the variable name with actual [53] or using the keyword without additional text [62].

Design and Discussion. See the corresponding paragraphs of Section 3.3.

Convention	Description
Actual	The name of the result is a case-insensitive match for actual
Actual Prefix	A proper prefix of the name of the result is a case-insensitive match for actual
Other	All other naming schemes for the result variable

3.5 Specifying Oracles

This class groups conventions on how to provide the value of the oracle provided to assertion methods.

Background. As introduced in Section 3.3, one recognized practice is to store the oracle in a local variable before passing it as an argument to an assertion method. By extension we can define alternatives for other common code elements that can hold a value, such as literals, fields, and method calls.

Design. The target element for this convention class is any *assertion* that involves an oracle–result pair. We include one alternative per syntactic category, plus a fall-through *Other* category to ensure the class is exhaustive. Each alternative can be detected by parsing the expression for the first argument in an assertion method.

Convention	Description
Variable	The first argument to the assertion method is parsed as a name expression
Literal	The first argument to the assertion method is parsed as a literal expression
Method Call	The first argument to the assertion method is parsed as a method call expression
Field Access	The first argument to the assertion method is parsed as a field access expression
Other	Another expression type is used as the oracle

Discussion. While some alternatives may appear exotic (e.g., *Field Access*), they are nevertheless justifiable in limited scopes (e.g., certain testing packages). Detecting this alternative may help developers identify little-known but desirable conventions in some parts of their test suite.

3.6 Supplying Failure Messages

This test convention class groups conventions regarding the use of the optional *message* argument in assertion methods.

Background. In JUnit, all assertion methods are overloaded to include both a basic version that verifies a predicate, as well as a version that takes in an additional *message* argument to output if the predicate is false (i.e., the test fails). Similarly, the JUnit API includes a method, fail, which immediately fails a test. This method is also available in two

versions that take in a message or not. This design of the JUnit API naturally induces two alternatives: whether or not to supply a test failure message. Some JUnit tutorials point out this feature [38, 53].

Design. The target element for this convention class is any *assertion*. We define an assertion to be any call to a JUnit *assertion method*, including calls to method fail.

Convention	Description
Has Failure Message	A call to a JUnit assertion method includes a message
No Failure Message	A call to a JUnit assertion method does not include a message

Discussion. The design of this convention class is dictated by an alternative offered by the JUnit API.

3.7 Assertion Density

This test convention class groups conventions regarding the number of assertions per test method.

Background. Expert advice usually calls for test that are focused [35] and test a single concern [41] (and e.g., [11, 12, 19, 62]). In practice this advice can be reflected by the number of assertions present in each test. The concept of *one assert per test (OAPT)*, in particular, generates much debate [55].

Design. The target element for this convention class is any test method. We designed this convention class by partitioning the possible range of number of assertions in a test into meaningful categories. The *No Assertion* alternative captures tests that do not use assertions due to an error, reliance on helper methods, or other reasons. The *One Assertion* alternative captures cases where the OAPT guideline is followed systematically. The *Few Assertions* alternative captures focused tests that do not strictly follow OAPT. The definition of this alternative as mapping to two or three assertions is arbitrary: in practice, it could be trivially modified to fit project practices. Finally, the *Many Assertions* alternative captures practices that deviate from common advice with tests that have more than a token number of assertions.

Convention	Description
No Assertion	The test method has no standard assertions
One Assertion	The test method has exactly one assertion
Few Assertions	The test method has two or three assertions
Many Assertions	The test method has more than three assertions

Discussion. The design of this convention class is more akin to a basic method classifier than a set of rules to check specific practices. For field use, we expect that the alternatives *No Assertion* and *Many Assertions* will be mostly useful to detect flaws and unorthodox implementations rather than verify a convention. The distinction between *One Assertion* and *Few Assertions* will also not be meaningful for projects that do not aim to strictly respect OAPT. For such projects the two alternatives would need to be merged. Despite these limitations, we include this convention class as a means to study the pervasiveness of strict OAPT practices in open-source projects. In an experimental context, occurrences of the *One Assertion* alternative can trivially be folded into the *Few Assertions* alternative post hoc.

3.8 Testing Exceptions

This test convention class groups conventions on how to test that an exception is thrown as expected.

Background. When a method is expected to throw an exception under certain conditions, tests can be written to ascertain this expectation. Over its different versions, JUnit has provided different mechanisms for asserting whether a method throws an exception of a given type. The *try/catch idiom* was used as early as JUnit3. With this idiom, a call to the focal method is placed in a `try` block whose `catch` clause defines a parameter of the type of the expected exception, and a call to `fail` is placed immediately after the call to the focal method.[5] JUnit4 introduced a different mechanism by adding an attribute `expected` to the `@Test` annotation used to mark a method as a unit test. The value of this attribute is a class literal referencing the class representing the type of the expected exception. Finally, JUnit5 homogenized the testing of exceptional behavior by introducing an assertion method for exceptions, `assertThrows`.

Design. The target element for this convention class is any test method that implements an exception testing mechanism. We define one alternative per mechanism, plus one alternative to represent mixed or ambiguous use of exception assertion mechanisms. Each mechanism can be detected by identifying its defining feature in the source code of the test.

Convention	Description
Expected Attribute	The method uses JUnit's `expected` attribute in the `@Test` annotation
AssertThrows	The test method calls JUnit's `assertThrows` method
Try Catch Idiom	The test method uses the `try-catch` idiom with a call to JUnit's `fail` method in the try block
Mixed Testing	The test method uses a mix of the other conventions in order to test exceptions

Discussion. The design of this convention class is dictated by alternatives offered by the JUnit API, so the definition of the alternatives fall naturally along those lines. The last category, *Mixed Testing*, is an artificial alternative to capture flawed or unorthodox exception testing. Although useful for detecting opportunities for improvement, we do not consider it a convention that one would aim to follow. Finally, our prototype implementation is limited to exception assertion mechanisms provided by JUnit. Test methods relying exclusively on third-party libraries for testing exceptions will not be considered as target elements for this convention.

3.9 Handling Exceptions

This class groups conventions guiding how to handle unexpected exceptional behavior (i.e., run-time errors) in the tests themselves.

Background. As any other piece of running code, executing tests can encounter unexpected problems: a null pointer dereference due to a bug, missing input data, etc. Different different strategies exist to handle such problems. First, the test code can be placed in a `try` block whose catch clause explicitly fails the test with a call to `fail()`. Second, the test method can have its definition extended to explicitly declare to throw some exceptions (which will propagate and fail the test). A third option, applicable only to unchecked exception types, is to do nothing, and let the exceptions propagate unannounced, which will also fail the test.

Design. The target element for this convention class is any test method that implements an exception handling mechanism. We define one alternative per detectable mechanism, plus one alternative to represent mixed or ambiguous use of exception assertion mechanisms. Each mechanism can be detected by identifying its defining feature in the source code of the test. It is not possible to detect the "do nothing" strategy because, by definition, it leaves no trace in

[5] If the call does not raise an exception, the test fails via the call to `fail()`; If the call raises an exception of a type not assignable to the type in the `catch` clause, the test fails via exception propagation.

the unit test and may be realized involuntarily. For this reason, the scope of this convention class is limited to *explicit* exception handling in tests.

Convention	Description
Fail in Catch Clause	A test method contains catch clauses with a call to a method named `fail`
Throws Exception	A test method declares to throw one or more exception types
Mixed Handling	A test method declares to throw one or more exception types and also contains catch clauses with a call to a method named `fail`

Discussion. Besides the issue with the "do nothing" strategy described in *Design*, above, another source of ambiguity for this convention class is Java's dual checked vs. unchecked exception handling mechanism. While explicitly handling an exception is mandatory for checked exceptions[6], the handling is optional, and usually omitted, for unchecked exceptions. The design of our convention class does not make this distinction mainly for conceptual reasons: the convention is about the *choice* of which exception handling strategy to employ, and not whether to employ one or not. Determining which exceptions can flow out of a block of code (and should therefore be handled or declared) is a challenging problem with imprecise solutions [48], and is outside the scope of this research.

3.10 Documenting Test Methods

This class groups conventions related to the documentation of test methods using Javadoc comments.

Background. JUnit tests are methods and, as such, can be documented with block comments, and in particular with block comments that follow the Javadoc conventions. It is also possible to explicitly link a test to its focal method using a `@link` or `@see` Javadoc tag.

Design. The target element for this convention class is any test method. We define as alternatives the different degrees of use of the Javadoc syntax.

Convention	Description
Has Linked Javadoc Comment	The test method has a Javadoc comment associated with the focal method using a `@link` or `@see` Javadoc tag
Has Javadoc Comment	The test method has an associated Javadoc comment not linked to the focal method
No Javadoc Comment	The test method does not have an associated Javadoc comment

Discussion. We limit the scope of this convention class to Javadoc comments, thus ignoring line comments and non-Javadoc block comments. The rationale for this decision is that Javadoc defines a universally-recognized format for systematically documenting code elements in Java, and thus we expect that projects that aim to systematically follow a documentation convention would use it. In addition, Javadoc comments explicitly target specific code elements and their use does not introduce any ambiguity about their target. In contrast, basic block comments and line comments may or may not be relevant to their nearby code element.

[6]Defined as a subtype of `Exception` that is not also a subtype of `RuntimeException`.

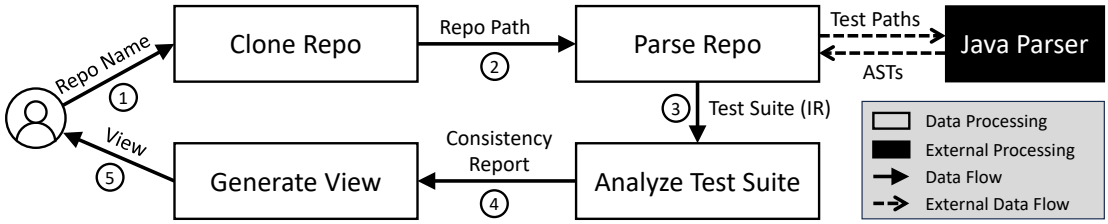


Fig. 2. Computing and viewing consistency information about Java test suites with Teslo and TestComet. The two data processing stages on the left correspond to Teslo and the two stages on the right correspond to TestComet.

4 SUPPORTING TEST CONVENTION CONSISTENCY ASSESSMENT IN PRACTICE

The metrics we introduced in Section 2.2 provide a technical means of reasoning about test convention consistency in the abstract. Providing an assessment of this consistency in practice requires supporting tools, and many design alternatives exists to realize such tools. We investigated the feasibility of supporting test convention consistency in practice by developing a prototype tool to:

(1) Compute test suite consistency for any Java project;
(2) Represent test convention classes and test conventions in a way that facilitates customizability;
(3) Display consistency scores in an understandable and actionable manner.

We designed and developed a suite of tools that consists of a static analyzer for computing consistency data (TestComet) and a web application for allowing users to obtain and view this data (Teslo).[7] Developing this tooling allows us to validate the practical viability of the idea and to identify the important design decisions necessary to realize it. We contribute these design decisions as a complement to the definition of the metrics and convention classes.

4.1 General Workflow

While TestComet can analyze any Java project, Teslo facilitates integration with GitHub. The usage scenario begins with a user entering the fully qualified name of a repository (i.e., `owner/repo`) in an input field of the web interface and clicking a button to launch the analysis (① in Figure 2). Teslo then clones the repository and provides its path as input to TestComet (②). TestComet then parses the repository to generate an intermediate representation (IR) of the test suite and provides this IR to an analysis module (③), which tallies the number of occurrences of each test convention in each test convention class in the form of a *Consistency Report*. This report is then made available to Teslo (④), which generates the interactive *Consistency View* and makes it available to users (⑤).

4.2 Analyzing Test Suite Consistency with TestComet

As illustrated in Figure 2, TestComet realizes two main functions: 1) to convert (or *parse*) a Java project into an *intermediate representation* (IR). Our test suite IR is a simplified version of an abstract syntax tree (AST) focused on code elements relevant to testing. Specifically, the IR can represent an instance of a test suite as a hierarchy of subtypes of `TestElement` that include `TestSuite`, `TestPackage`, `TestClass`, `TestMethod`, and `Assertion` (each element type aggregating one or more instances of the next).

[7]TestComet stands for **Test Co**nsistency **met**rics and Teslo for **Tes**t **lo**ve. At the time of submission, TestComet and Teslo are not open-source for intellectual property protection reasons.

The first task of TestComet is to identify the set of files that belong to the repository's test suite. This classification can easily be parameterized. In our version, we considered a file to be part of a test suite if it 1) has the .java extension, 2) is located within manually-validated test directories (e.g., src/test/java), and 3) includes the JUnit import pattern [64].[8].

For each test file in the test suite, TestComet uses the JavaParser [22] library to generate an AST from its source code. We traverse the AST to identify all of the test methods in each test class, as determined by the JUnit's annotations. For each test method, the tool then identifies the *assertions* it contains. Determining precisely what constitutes an assertion is a problem that has no precise solution in the general case since developers could technically implement their own idiosyncratic assertion methods and idioms. We solved the problem pragmatically by considering any call to a method that begins with the string "assert" to be a potential assertion. While there is a small risk of false positive, the major advantage of this strategy is that it allows us to detect most assertions from third-party matcher frameworks as well as sensibly-named user-defined assertion methods. A study of over 4,500 open-source Java projects revealed that developers often use matcher frameworks alongside JUnit [65]. The JUnit team recommends using Hamcrest,[9] AssertJ,[10] or Truth,[11] each of which rely on a base method called assertThat.

Once an instance of TestSuite has been constructed by the parser, TestComet can determine how often and where each test convention occurs. This analysis relies on a configurable list of *test convention classes*, such as those detailed in the catalog in Section 3. In our implementation, each test convention class is supported by one (Java) class with a single responsibility: to determine whether an input TestElement constitutes an occurrence of a convention in the convention class. For this purpose, test convention classes aggregate a *convention detector*. Convention detectors visit the test suite IR, identifying the test elements that could be an occurrence of a convention in the convention class. For example, the convention detector for Naming Test Classes extracts all the SimpleName nodes corresponding to test class names. For each candidate element, the convention detector calls an isOccurrence method for each convention alternative in the class. This delegation reduces coupling by separating concerns, facilitating customizations. Developers can reuse convention detectors for convention classes with the same candidate element types. Creating new test convention classes is straightforward, as developers can mix and match test conventions and convention detectors.

TestComet 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 consistency metrics.

4.3 Exploring Test Suite Consistency with Teslo

Developers can use the Teslo web application to analyze the consistency of any JUnit test suite that is part of a GitHub repository they have access to. To do so, they enter the fully qualified name of a repository (i.e., owner/repo) in the input field and click *Analyze*, as depicted in Figure 3, Frame A. The application then takes the input repository name and clones the corresponding repository locally. The application then provides the path of the cloned repository to TestComet, which produces a *Consistency Report* (see Section 4.2).

Once available, Teslo uses the consistency report to generate a *Consistency View*. We sought to display the consistency scores in an understandable and actionable manner. To that end, and inspired by test coverage views,[12] we made the consistency view hierarchical and interactive. The consistency view initially shows a high-level summary of the test

[8]The file must include lines that start with import org.junit.jupiter.
[9]http://hamcrest.org/
[10]https://assertj.github.io/doc/
[11]https://truth.dev/
[12]Such as EclEmma's, https://www.eclemma.org/userdoc/coverageview.html

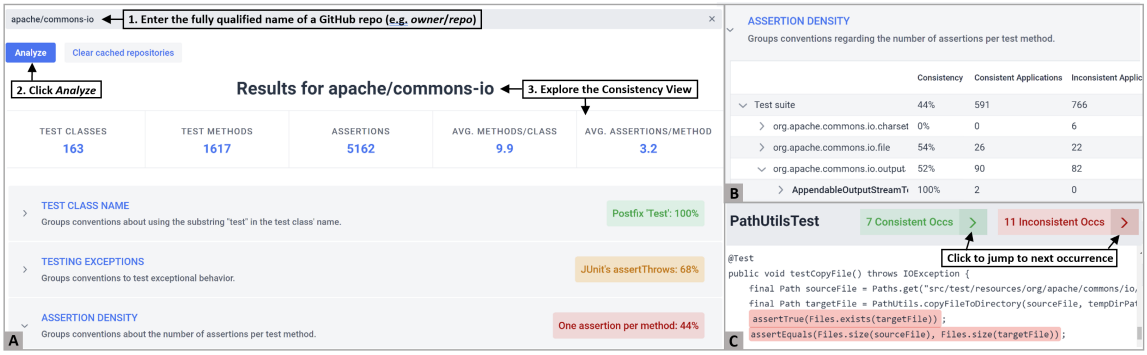


Fig. 3. Teslo UI. Frame A (left) illustrates the initial, test-suite-level view of the consistency scores. Frame B (top right) shows how users can drill down to view the consistency scores at the test package, class, or method level. Frame C (bottom right) presents the most detailed view—the annotated code dialog that appears when clicking a test class. It allows users to jump through the consistent and inconsistent occurrences in the test class.

suite's consistency using the accuracy-based metric (see Section 2.2).[13] 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 consistency score of the convention class at the test suite level. The color of the badge varies with the consistency level: green (Consistency$_A$ > 75%), orange (Consistency$_A$ > 50% and ≤ 75%), and red (Consistency$_A$ ≤ 50%).

Depending on their information needs, users 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 3, Frame B). This feature allows users 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 detailed view, users can click on any test class to open a dialog that displays its source code. As demonstrated in Figure 3, 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 users understand the contexts in which they and their team apply alternative conventions. Users can then decide which alternative(s) to adopt. They can also assess whether deviations from the target convention are justified or unnecessary.

4.4 Related Tools

The closest alternatives to TestComet are automated static analysis tools (ASATs) such as FindBugs,[14] PMD,[15] and Checkstyle.[16] ASATs detect common programming errors and convention 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 [23] and poor customizability [3, 63]. We sought a solution that would address, to the extent possible, these design challenges.

[13]There is no fundamental limitation that would prevent displaying the entropy-based metric as well. Because our development time is bounded, and we experimented with the entropy-based metric extensively through case studies (see Section 5), we focused our efforts on presenting the results of the accuracy-based metric in Teslo.
[14]http://findbugs.sourceforge.net
[15]https://pmd.github.io
[16]https://checkstyle.sourceforge.io

The main distinction between TestComet/Teslo and traditional ASATs is that ASATs are prescriptive and flag any deviation from a single convention, whereas our analysis framework not only detects a range of alternative conventions, but also leverages this information to offer a synthesis of a test suite's consistency as a component of its quality. Code conventions can emerge naturally over time as the result of various local decisions made by different developers [1]. Without tool support, it can be 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.

ASATs support customizations in terms of enabling and disabling rules and adding custom rules. However, developers find existing tools challenging to configure [23]. They rarely alter the default configurations, adopting ASATs as-is instead [3, 63]. This phenomenon is not due to a lack of demand, as developers deem customizability an essential feature of ASATs [23]. As such, we sought to represent convention classes and conventions so as to facilitate customizability.

ASAT reports typically comprise lengthy lists of warnings. Previous work suggests that this output is difficult to understand. For instance, Johnson et al. [23] 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 [50]. As a result, various studies found that developers only fix a small portion of ASAT warnings [32, 34]. 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.

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. A comprehensive study of such factors is outside the scope of this article. However, as a first step, we provide evidence of the suitability of our proposed consistency metrics and tool infrastructure to reason about test suite evolution. Our goal was to explore the potential of test-suite consistency analysis for macroscopic quality assessment of evolving test suites. We sought to answer the following research questions:

RQ1 How do the metrics compare across convention classes?
RQ2 What is the consistency of different high-profile projects?
RQ3 Which contextual factors can affect the consistency metrics and how?
RQ4 What types of actionable insights are supported by an analysis of test suite consistency?

We conducted a multi-case study of open-source JUnit test suites hosted on GitHub. In our study design, we define a *case* as the *test suite* for a *Java project*, pragmatically equated to a repository on GitHub. We elected to study 20 distinct projects to strike a balance between breadth (gaining insights from different contexts) and depth (the extent to which we can account for the individual characteristics of each case).

Data Artifact. This article is complemented by an on-line archive that includes the datasets underlying the analyses and visualizations described in this section [49].

5.1 Projects Studied

Our target population is *active* and *collaborative* Java test suites hosted on GitHub that use the JUnit5 framework. We limit the scope of our study to JUnit5 to focus on well-maintained test suites (JUnit5 is more than six years old) and to avoid the imprecision and ambiguity of analyzing multiple overlapping frameworks and versions.

We queried the GitHub API[17] to retrieve an initial *candidate list*. Although we are interested in test suites, the GitHub API does not support querying for such a construct. Instead, we used the API to identify a set of candidate Java repositories from which we could extract test suites. GitHub hosts over 12 000 000 public Java repositories (as of 2023-12-03), most of which are not relevant to our study as they are personal projects or inactive [25]. To ensure that the candidate repositories are appropriate, we implemented Kalliamvakou et al.'s [25] selection strategies in the form of the following inclusion criteria:

- **Active**: the repository must have at least one commit on the default branch in the past six months (since 2023-05-12) 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 [21] definition of a mature repository: the repository is at least two years old (created before 2021-11-12) and has 100 or more commits on the default branch.
- **Collaborative**: the repository must have at least three unique human contributors given that 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 used this threshold to filter out repositories without a minimum of community recognition.

We applied these inclusion criteria through query parameters in the GitHub API and by further analyzing the response data. We performed the query on November 12th, 2023, and it produced 4960 candidate repositories.

Next, we constructed a *sampling frame* by filtering the candidate list to only include repositories with a test suite. We used the same three criteria as TestComet to determine whether a file is part of a test suite (see Section 4.2). We retrieved each repository's tar archive from GitHub and extracted all files with the .java extension. We then identified within the directory structure the location of the test suite. Most projects used a conventional structure, e.g., placing tests within a src/test/java directory.[18] We extracted files having JUnit5 import patterns.[19] We only included candidate repositories whose test suite had at least 100 test classes to ensure they had sufficient test-related commits to analyze.

The final sampling frame comprised 444 projects. We used this sampling frame as the basis for selecting our target projects. To focus on high-visibility and high-impact projects, we drew from our sampling frame in decreasing order of GitHub stars, excluding any project that was not a software system, whose documentation was not in English, which relied heavily on a testing framework other than JUnit, which was conceptually too similar to an already-selected system, or whose organization caused a threat to validity. With this process we rejected ten projects, resulting in 20 projects.[20]

Table 2 provides the list of selected projects. The first two columns describes the full project name and the first six digits of the commit hash of the last version we analyzed. From this information it is possible to retrieve the exact state of the projects from GitHub.[21] Henceforth, we refer to projects by repository name only, e.g., Spring-boot. The other

[17] https://docs.github.com/en/rest
[18] We initially included each file with "test" in the file path for the automated extraction scripts, then manually validated and refined the directory filters.
[19] Lines starting with import org.junit.jupiter
[20] We rejected three projects that were tutorial or demonstration code, one project with non-English documentation, one project with a large number of duplicated files, two projects using other testing frameworks, and two projects that were variants of, or add-ons to, already selected projects.
[21] Using the url https://github.com/<owner/repo>/tree/<commit>

Table 2. Projects Studied. *Hash* represents the first six digits of the commit hash of the last version we analyzed. *Classes* and *Methods* indicate the number of test classes and methods we detected. *Age (days)* indicates the number of days between the first and last modifications of the test suite and *Commits* indicate the number of commits that modified the test suite during that period. *Contr.* indicates the number of unique contributors to the test suite. *Conv.* indicates whether a systematic search revealed the presence of a guide prescribing test conventions for the project: ○None found; ◐Guide with no relevant convention; ●Guide with relevant conventions.

		Test Suite Properties (JUnit5 only)					
Project (owner/repo)	**Hash**	**Classes**	**Methods**	**Age (days)**	**Commits**	**Contr.**	**Conv.**
deeplearning4j/deeplearning4j	`61c8cc`	657	4686	950	196	3	○
apache/dolphinscheduler	`65a7c7`	451	1970	744	662	135	◐
apache/dubbo	`2ca55a`	799	3710	1761	1006	180	○
apache/flink	`3dd984`	1571	9334	822	3487	304	○
apache/hadoop	`a32097`	181	1015	1667	889	93	●
skylot/jadx	`e6d896`	534	756	1674	401	19	○
apache/kafka	`7c562a`	744	7155	1156	1914	273	●
mybatis/mybatis-3	`311575`	329	1656	1768	186	6	◐
neo4j/neo4j	`60f235`	1828	13 149	1981	7512	104	◐
netty/netty	`285ba7`	530	4877	1068	435	96	●
pinpoint-apm/pinpoint	`9b9dff`	1056	3189	496	302	16	○
quarkusio/quarkus	`2f2ce0`	4182	10 546	1752	5346	48	○
SeleniumHQ/selenium	`4d1b00`	300	2777	510	266	30	○
apache/shardingsphere	`f8a420`	1377	6054	258	1309	70	●
apache/skywalking	`019c6f`	202	532	1334	342	75	○
spring-projects/spring-boot	`e4a0e9`	2127	14 149	2299	5617	26	○
spring-projects/spring-framework	`7f615f`	2139	19 763	2685	5320	197	●
thingsboard/thingsboard	`afa54e`	123	726	970	452	12	○
zaproxy/zaproxy	`8f3932`	184	2265	1393	146	4	◐
apache/zookeeper	`75d0a0`	325	1626	1220	132	54	●

columns describe properties relative to the projects' test suite: the number of test classes (*Classes*), of test methods (*Methods*), of days between the first and last modifications of the test suite (*Age (days)*), of commits that modified the test suite (*Commits*), and of unique contributors to the test suite (*Contr.*).[22] The last column, *Conv.*, indicates whether our systematic search revealed the presence of a guide prescribing test conventions for the project (see Section 5.2).

In each project, for each test-related commit (i.e., that modified files included in the test suite), we computed the accuracy-based and entropy-based consistency of all convention classes. When a test suite did not contain an instance of any convention in a convention class, we considered the consistency *undefined* (as opposed to, e.g., 0 or 1), to avoid generating spurious extreme values. As a result, we obtained 400 data series about the evolution of consistency (two metrics × twenty projects × ten convention classes). We analyzed these data series to assess how the consistency metrics can help monitor the quality of test suites.

One potential factor that can affect the consistency metrics (RQ3) is the impact of the involvement of different types of contributors to a project, such as core vs. peripheral developers. Analyzing contributor involvement requires associating test-related commits to specific developers as reliably as possible. However, developers can use different email addresses and variations of their names when authoring commits. We thus performed name and email address

[22]The properties exclude contributions to the test suite before the adoption of JUnit5. The number of test classes and methods reflect the state of the test suite at the latest commit.

unification to link commits to developers more accurately. We considered two {name, email address} combinations to belong to the same author if they had the same email address or name. However, we only matched on the author's name if the name (1) was not "no author" and (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>") [25]. The *Contr.* column of Table 2 accounts for this unification process.

5.2 Test Convention Guides

An important contextual attribute for each case is whether the project *prescribes* test conventions or not. This distinction is important when interpreting to what extent different conventions are respected and thus the level of consistency we observe. We investigated whether each project prescribes test conventions for contributors. As the organization of developer documentation is not standardized, this investigation inevitably followed a heuristic process. We designed a systematic search process for a *test convention guide* for a given GitHub repository, and applied it to all our target projects. We conducted the search as follows:

(1) Inspect the README and CONTRIBUTING files at the root of the repository.[23] While the README is nominally intended for users of the project (as opposed to contributors) [14], it can also provide an overview of the repository, including links to documents for contributors. The CONTRIBUTING file is explicitly designated by GitHub as a resource for describing "guidelines for contributors" [15], and thus the logical place to look for testing guidelines. From those two pages, we transitively followed any link to resources for developers, searching for the individual keywords: test, convention, style, guide, and format. We followed all potentially-relevant links, including to resources external to GitHub (e.g., project website or wiki).
(2) We conducted a repository-wide search for any documentation page with the string "test" in its path.[24] This step ensures that we had not missed a link-path to a relevant resource during Step 1. We consulted each result file and searched for the same keywords as for Step 1.
(3) We inspected the GitHub wiki's table of contents for relevant pages.

We scoped our search to human-targeted documents. For this reason, we did not target our query to directly include tool configurations files (e.g., style.xml, the configuration file for Checkstyle). We reasoned that to be effective as prescriptive conventions, any style configuration file targeting unit tests would need to at least be mentioned in developer documentation. If a configuration file was referred to in developer documentation, we considered it for inclusion in the search.

As the search process is heuristic, we acknowledge that other strategies are possible to refine it. Nevertheless, if a project intends to require contributors to follow conventions, a natural imperative is that these conventions be conspicuous. The fact that the search process surfaced test conventions guides for some projects with very different organization of their documentation testifies to its suitability. Although it is possible that we missed a convention guide for a project, this guide would be hard enough to find to put its effectiveness in question.

Once we identified a guide for a project, we reviewed its conventions and determined if it included a convention from our catalog (see Section 3). If the convention is supported by TestComet, we considered it *relevant* to the research. Table 2 summarizes to what extent each of our target project's test conventions were documented, and Table 3 provides the details of the conventions.

[23] Considering any extension. We found files with extensions .txt, .adoc, and .md.
[24] For a given REPO, we used the search query repo:REPO path:/^.*test.*.(adoc|md)$/.

Table 3. Documented Relevant Conventions

Project	Convention Class	Requirement
Hadoop	Naming Test Methods	"Define methods within your class whose names begin with test"
	Supplying Failure Message	"add meaningful messages to the assert statement"
Kafka	Naming Test Methods	"test method name begins with "check""
	Naming Test Classes	"class name begins with "Check""
Netty	Naming Test Classes	A tool verifies that the names of test classes match the regular expression [A-Z][A-Za-z\d]*(Test|Benchmark|Microbenchmark)[0-9]*
Shardingsphere	Naming Oracles	Expected Prefix
	Naming Results	Actual Prefix
	Documenting Test Methods	No Javadoc Comment
Spring-framework	Naming Test Classes	Test Suffix
Zookeeper	Naming Test Methods	"Define methods within your class whose names begin with test"

However, we can readily draw two important conclusions from these results. First, we observe little to no consistency between projects within an organization. As a clear example, Spring-framework has a *Code Style* page with test naming conventions on its wiki, but Spring-boot simply has a *Code Conventions and Housekeeping* section in its CONTRIBUTING.md, with nothing about testing. Likewise, although eight of the projects in our list are Apache projects, they do not overtly share a common approach to test conventions. Second, for the projects that do have a test convention guide, we observed a considerable variety of ways in which these guides are organized and disseminated. Examples include:

- A special document unit-test.md deep within the docs directory tree (dolphinscheduler);
- A page on an external wiki (hadoop);
- A separate README file in the tests folder (kafka);
- A page on the GitHub wiki (mybatis-3);
- A page on an external project website (neo4j);

Hence, we conclude that the projects in our list exhibit largely independent approaches to documenting test conventions.

5.3 Analysis of Consistency Metrics

The two metrics we introduced in Section 2.2, accuracy-based and entropy-based, provide two complementary but *consistent* ways to synthesize the state of a test suite. Before leveraging the consistency metrics to reason about test suite evolution, we provide a technical analysis of the relation between the two metrics, followed by an illustration of their range on our target projects.

Technical Analysis. From the definition of the two consistency metrics, we can derive theoretical relations between the values that they can take for the same test suite. Because the entropy-based metric can be interpreted as a weighted mean of the probabilities in a distribution, whereas the accuracy-based metric is equal to the highest probability, it follows that, for any test convention class TCC, $\text{Consistency}_E(TCC) \leq \text{Consistency}_A(TCC)$.

If we fix the number of conventions in a class, it is possible to bound the possible values of entropy-based consistency for any value of accuracy-based consistency. Figure 4 shows the possible values that both metrics can simultaneously

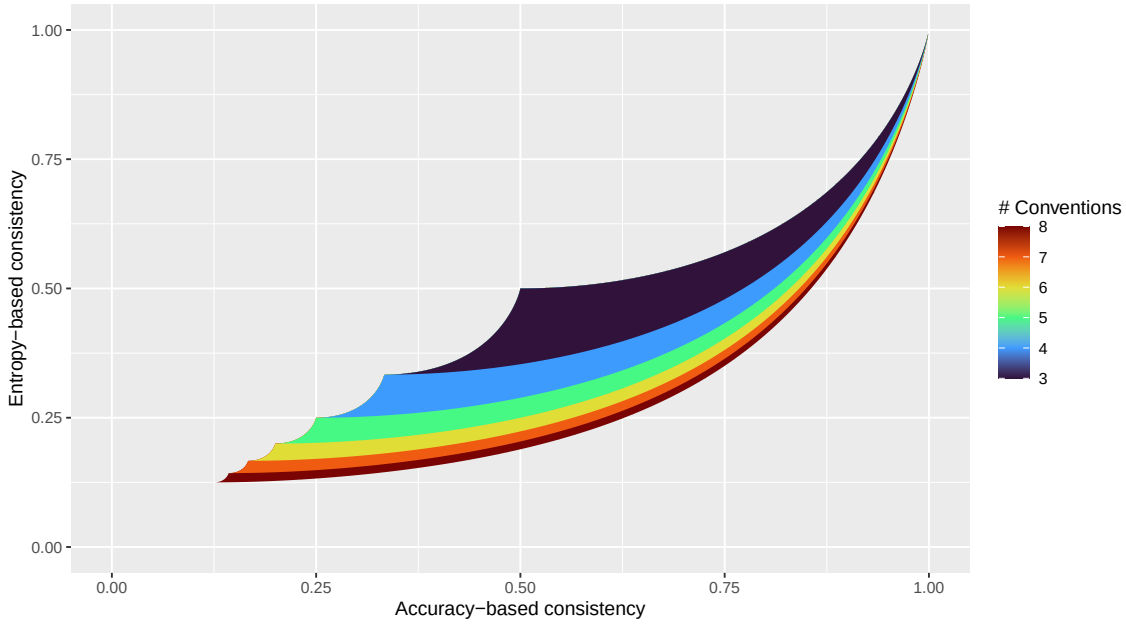


Fig. 4. Simultaneous range of both metrics given the number of conventions. The range for a number of conventions overlaps with the range for lower numbers of conventions.

take for the same test suite, based on the minimum number of conventions in a class. The topmost and largest colored region corresponds to three conventions. If there are only two conventions, the entropy-based and accuracy-based metrics are related by a deterministic function, which corresponds to the line forming the top edge of the area, from 0.5 to 1.0. As an example of interpretation of the figure for a convention class with three conventions (the darkest region), if we observe the value of the accuracy-based metric at 0.50, we see that we can expect the value of the entropy based metric to be between 0.35 and 0.50. The region for a higher number of conventions include all regions for lower numbers of conventions. For example, the possible values for a convention class with eight conventions include all areas, not just the slim dark red area.

Figure 4 shows only the area for a number of conventions up to eight. As the number of conventions continues to grow, the range of possible values would fill the entire lower part of the plot. That is, for any value of accuracy-based consistency, the entropy-based consistency can be arbitrarily close to zero if the number of conventions is sufficiently large.

This theoretical comparison of the two metrics demonstrates that the entropy-based consistency penalizes more strongly small deviations from the most common convention: Entropy-based values are more discriminating than accuracy-based values when there is a clearly dominating convention (i.e., consistency close to 1.0). In contrast, entropy is less discriminating than accuracy when all conventions are used almost as often as each other (i.e., consistency close to $1/N$ when N conventions are used).

Ranges in Practice. We used the consistency values of all convention classes for the latest commit of all projects to compare the two metrics in practice, shown in Figure 5. Each line represents the consistency of a project's test suite

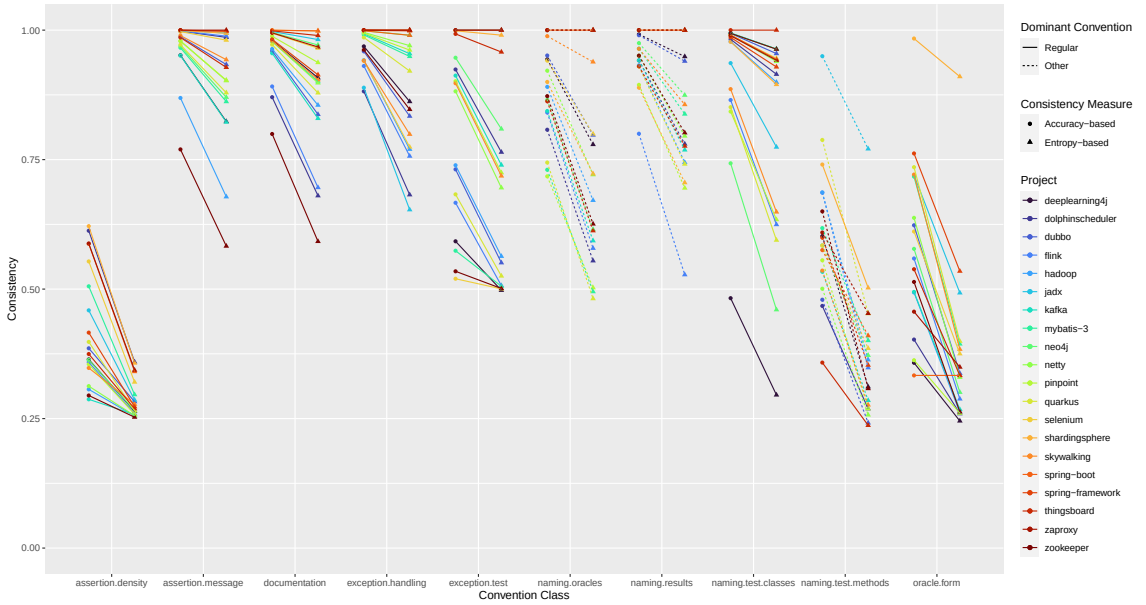


Fig. 5. Consistency of test suites at the latest commit. For each convention class (x axis), the line segment goes from the accuracy-based consistency (left) to the entropy-based consistency (right). The horizontal distance of each line is constant. Dashed lines indicate that the most common convention was the "catch-all" convention.

(color) for a convention class (x axis). The left end of the line shows the accuracy-based value and the right end shows the entropy-based value. Thus, the vertical distance represents differences between the two values. The horizontal distance between the metrics carries no information and is constant for all pairs of values. Dashed lines indicate cases where the most common (i.e., *dominant*) convention is the catch-all one (e.g., *Other*, *Mixed*).

The figure conforms to the property highlighted in our technical analysis in that all lines are constant or decreasing from left to right, indicating that entropy-based consistency is never higher than accuracy-based consistency. The lines also show the lower discriminating power of accuracy-based consistency compared to entropy-based consistency for consistent convention classes, and vice versa for inconsistent convention classes.

The lines in the figure seldom cross each other, which suggests a strong monotonic (although not linear) correlation between both metrics within each convention class. Spearman's rank correlation confirms this observation: $\rho > 0.986$ for all convention classes, except for Assertion Density ($\rho = 0.929$), Specifying Oracles ($\rho = 0.818$), and Naming Test Methods ($\rho = 0.803$). Nevertheless, the presence of crossings indicates that entropy-based consistency can reveal information about non-dominant conventions that the accuracy-based consistency hides. The most striking example is the consistency of Spring-boot for the Specifying Oracles convention class. Its accuracy-based value is the lowest among all projects, but its entropy-based value ranks eleventh out of the 20 projects. This is due to Spring-boot showing no dominant convention, but alternating between only three of the five conventions in this class.

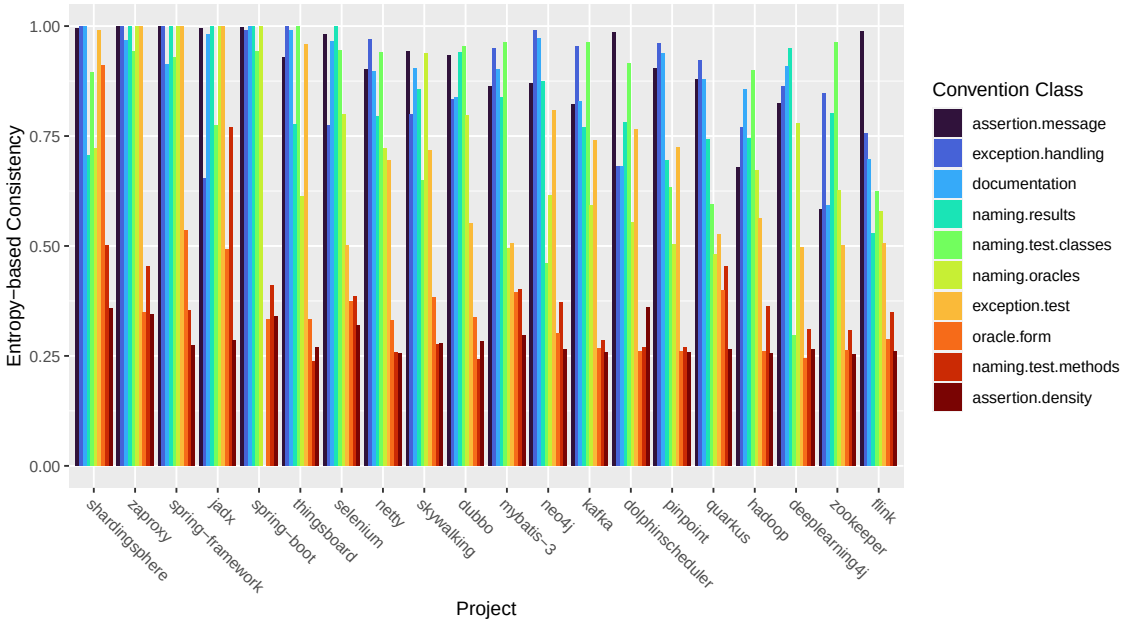


Fig. 6. Entropy-based consistency of each project at the latest commit. Projects and convention classes are ordered by average consistency. One value is missing for Spring-boot as it did not include any instance of a convention in the Testing Exceptions class.

RQ1: How do the metrics compare across convention classes? The entropy-based and accuracy-based metrics are consistent, but complementary. Entropy-based scores emphasize the negative impact of occasional deviations from a dominant convention. In contrast, accuracy-based scores emphasize the most popular convention in convention classes that are less consistent. In practice, both metrics are strongly positively correlated.

Figure 5 also provides an overview of the typical consistency values for different classes. Some classes (e.g., Handling Exceptions) are consistent for all projects, whereas others (e.g., Assertion Density) are mostly inconsistent. In particular, although formats for naming test methods are often discussed by experts (see Section 3.1), this convention class is largely inconsistent across projects. Furthermore, for 17 of the 20 projects, the dominant convention is the catch-all *No Focal Method*.[25] This failure to follow a consistent convention indicates a potential divergence between testing style guides and development concerns.

Figure 6 compares the entropy-based consistency of all projects at the latest commit. To improve readability, projects are ordered by average consistency across all convention classes, and convention classes are ordered by average consistency across all projects. Despite general trends across convention classes, each project struggles with different convention classes. For example, the most consistent projects overall, Shardingshpere and Zaproxy, are inconsistent in Naming Results and Specifying Oracles, respectively, relative to other top projects. Conversely, the generally inconsistent projects Flink and Zookeeper are among the most consistent in the convention classes Supplying Failure Message and Naming Test Classes, respectively. These differences show the value of studying consistency within different convention classes, rather than aggregating values into a single score per test suite.

[25]The dominance of *No Focal Method* cannot be attributed solely to the high number of conventions in the class. For most projects, this convention is used more often than all other conventions combined, as shown by the accuracy-based consistency values above 0.5.

RQ2: What is the consistency of different high-profile projects? Consistency varies across the 20 open source projects and across convention classes. Some convention classes, such as Assertion Density, are inconsistent for most projects, whereas others are more consistent, such as Supplying Failure Message. Even the most consistent projects fail to maintain a high consistency across all convention classes.

5.4 Analysis of Global Trends in Test Suite Consistency

We investigated trends in the evolution of all projects across all convention classes. For this analysis, we focused on the entropy-based metric, as it is sensitive to variations in non-dominant conventions. First, we evaluated the impact of including a catch-all convention in the design of a convention class, as it can pose a threat to the construct validity if this convention dominates others. Second, we investigated the impact on consistency of different contributors' involvement in a project. We hypothesized that peripheral contributors introduce inconsistencies more often than core contributors. The objective of this investigation is to understand how consistency metrics can support an analysis of test suite quality.

Impact of the Catch-All Convention. The design of a convention class can affect the consistency values for the targeted aspect of the test suite (see Section 2.4). Grouping many specific variations into a single broad convention can artificially increase consistency. When a convention class is open-ended, e.g., Naming Test Methods, a practical decision is to create a catch-all convention that, in effect, groups all possible conventions not explicitly covered by the other conventions in the class. The impact of this decision is limited if the frequency of the catch-all convention is low. However, if it becomes the dominant convention in a class, the test suite may have a deceptively high consistency value despite not following any fixed convention.

We sought to assess the magnitude of this phenomenon in practice for our experimental catalog of conventions. Two convention classes, Naming Oracles and Naming Results, are problematic in this regard. For both convention classes, in all but one project (Spring-boot) the *Other* convention dominates and consistency values are high (median of 0.778). These high values for the *Other* convention suggest that the convention class should be refined to properly capture consistency. The catch-all convention of Naming Test Methods, i.e., *No Focal Method*, is also the dominant one for all but three projects (Dolphinscheduler, Shardingsphere, and Thingsboard), but because the consistency values are lower (median of 0.353), the threat of inducing an inaccurate assessment of consistency is not as important.

In addition to providing a false sense of consistency, convention classes where the catch-all convention is dominant can inverse the meaning of consistency changes. For example, a commit that adds instances of the second most frequent convention (i.e., the most frequent regular convention) would decrease the consistency value. Thus, changes in the consistency of a convention class dominated by the catch-all convention should be interpreted with care.

Impact of Contributor Involvement. Developers that only occasionally contribute code to a repository (i.e., *peripheral* developers) may not be as acutely aware of testing conventions as the *core* developers of the project, especially when those conventions are not well documented (as is the case for most of the studied projects, see Section 5.2). Thus, we can expect peripheral developers to degrade the consistency of a test suite more often than core developers. We applied our consistency metrics to test this hypothesis.

For each commit and each convention class, we considered a binary variable indicating whether the commit degraded consistency. We did not consider the magnitude of the change in consistency, as the magnitude is largely influenced by the number of instances of the convention class prior to the commit. When there are already many occurrences of conventions in a convention class, adding or modifying a fixed number of occurrences will have a proportionately

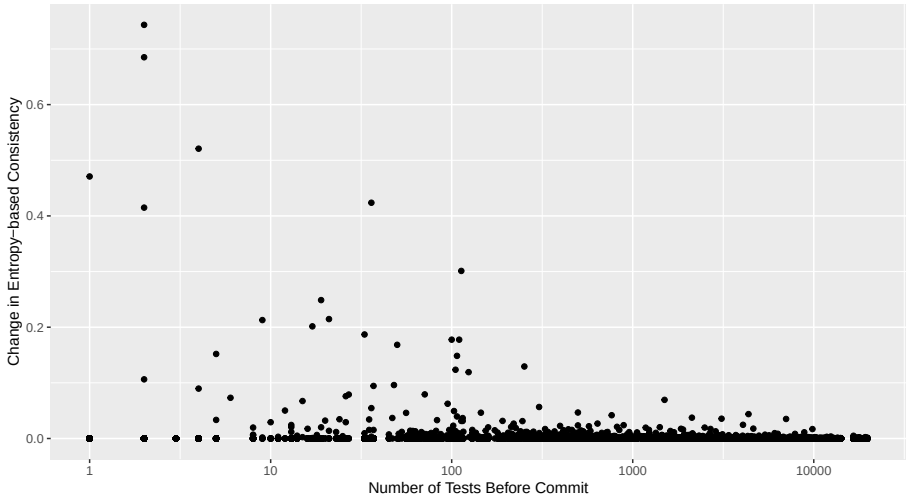


Fig. 7. Comparison of the number of tests prior to a commit with the absolute change in entropy-based consistency for Naming Test Methods caused by the commit. The consistency is only evaluated for Naming Test Methods. All commits from all projects are represented. The x axis is on a log scale.

smaller impact than when the number of occurrences is small. Figure 7 illustrates this relation by comparing, for Naming Test Methods, the absolute change in consistency with the prior number of tests (for Naming Test Methods, each test corresponds to exactly one instance of the convention class).

We also operationalized contributor involvement as a binary variable, *core* vs. *peripheral*, based on *code ownership*. A contributor owns a line of code if they made the last modification of that line of code. Based on Mockus et al.'s observations [39], we classified the top contributors who collectively own at least 80% of all lines of code as the core contributors, and others as peripheral.

Table 4 shows the p-values of one-tailed Fisher's exact tests comparing the proportion of consistency-degrading commits from core vs. peripheral contributors for all projects and all convention classes. Some tests could not be performed because our criterion identified no peripheral contributor (e.g., Deeplearning4j) or because there were no or only consistency-degrading commits. Asterisks (*) mark p-values below the 0.05 significance level. These should be considered carefully, however, as the high number of tests increases the probability of rejecting the null hypothesis (that there is no differences between the two groups). After applying the Bonferroni correction to the α level, only two test outcomes remain statistically significant.

Across all tests, few produce a p-value lower than our α level, even without the Bonferroni correction. Furthermore, there is no clear trend among convention classes or projects, except for Spring-boot for which all but one of the test outcome are below the uncorrected α level of 0.05.[26] Thus, we can hypothesize that peripheral contributors may

[26]We experimented with variations of the criterion for distinguishing core contributors, and observed similar conclusions. For example, increasing the cumulative ownership threshold to 90% made the test outcomes significant, after Bonferroni correction, for Assertion Density for Shardingsphere and Spring-boot, but not for Documenting Test Methods. Decreasing the threshold to 70% resulted in seven of the test outcomes becoming significant after correction. Changing the criterion entirely, to consider any contributor with at least three (resp. ten) commits as core, generated three (resp. two) significant tests after correction. Excluding commits that did not modify the distributions of conventions to remove a possible confounding factor identified the same two conditions as significant under the corrected α level as in Table 4. In all variations, all significant tests after correction were at least significant before correction with the original parameters (i.e., marked with an asterisk in Table 4).

Table 4. P-values of one-tailed Fisher's exact tests comparing the proportion of consistency-degrading commits by core vs. peripheral contributors. Asterisks (*) indicate significant results at the uncorrected 0.05 α level, and **bold font** indicates significant results after applying the Bonferroni correction. "N/A" values indicate that the test could not be performed due to one dimension having no instance.

Project	Naming Test Methods	Naming Test Classes	Naming Oracles	Naming Results	Supplying Failure Message	Assertion Density	Testing Exceptions	Handling Exceptions	Documenting Test Methods	Specifying Oracles
Deeplea.	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Dolphins.	0.9999	1.0000	0.9978	0.9971	0.9996	0.9991	1.0000	0.9994	1.0000	0.7617
Dubbo	0.0234 *	0.9988	0.9797	0.9641	0.9991	0.9440	0.9984	0.8611	0.9998	0.3688
Flink	0.9983	0.7003	0.0956	0.0229 *	0.6686	0.9991	0.1384	0.5806	0.9995	0.0004 *
Hadoop	0.3213	0.5490	0.7353	1.0000	0.7160	0.4932	0.5612	0.2303	0.5803	0.7014
Jadx	0.0931	0.0100 *	N/A	N/A	0.4253	0.0016 *	N/A	0.6992	**9.1e-5** *	0.1133
Kafka	0.0373 *	0.8309	0.9300	0.5457	0.2580	0.3025	0.0190 *	0.4408	0.9866	0.4038
Mybati.	1.0000	N/A	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Neo4j	0.3773	0.9731	0.7718	0.3314	0.9972	0.4690	0.6070	0.9735	0.9066	0.5631
Netty	0.1877	0.9558	0.9874	0.9929	0.7256	0.0003 *	0.9993	0.9888	0.3151	0.0018 *
Pinpoint	0.5173	0.3919	0.4089	0.5510	0.2609	0.2985	0.4129	0.1511	0.4954	0.6954
Quarkus	0.4001	0.2148	0.0518	0.2128	0.0350 *	0.7498	0.9792	0.2739	**9.0e-6** *	0.1946
Selenium	0.6692	0.3991	0.4107	N/A	0.0012 *	0.9842	0.4054	0.5589	0.0100 *	0.4107
Sharding.	0.0605	0.9890	0.1190	1.0000	0.9996	0.0018 *	1.0000	0.9788	0.9912	0.4529
Skywalk.	0.5571	0.9489	1.0000	1.0000	1.0000	0.9558	1.0000	1.0000	1.0000	0.0731
S.-boot	0.0266 *	0.0266 *	N/A	N/A	0.0006 *	0.0006 *	N/A	0.0419 *	0.0302 *	0.3514
S.-fram.	0.0009 *	0.8004	1.0000	N/A	0.6599	0.0024 *	0.1367	0.9935	0.8756	0.3170
Thingsb.	0.9191	1.0000	1.0000	1.0000	1.0000	0.6748	1.0000	N/A	1.0000	1.0000
Zaproxy	0.9959	0.9929	N/A	N/A	N/A	0.5213	N/A	N/A	0.9004	0.3918
Zookeep.	0.3116	1.0000	0.9630	1.0000	0.8986	0.7518	0.9391	0.9993	0.8066	0.1883

introduce inconsistencies in specific cases, possibly more often in Spring-boot, but we cannot conclude that there is a systematic bias, even within any one of the convention classes.

One factor that could explain these negative results is the inaccuracies in the commit authorship information. Practices such as squashing commits from pull requests or merging pull requests without retaining the original author's credit can eliminate information about the contribution of peripheral contributors. Pull request reviews can also introduce inaccuracies in the data set: a change suggested by a core contributor to increase consistency, but implemented by the peripheral contributor, would misrepresent the role of contributors in managing the test suite's consistency.

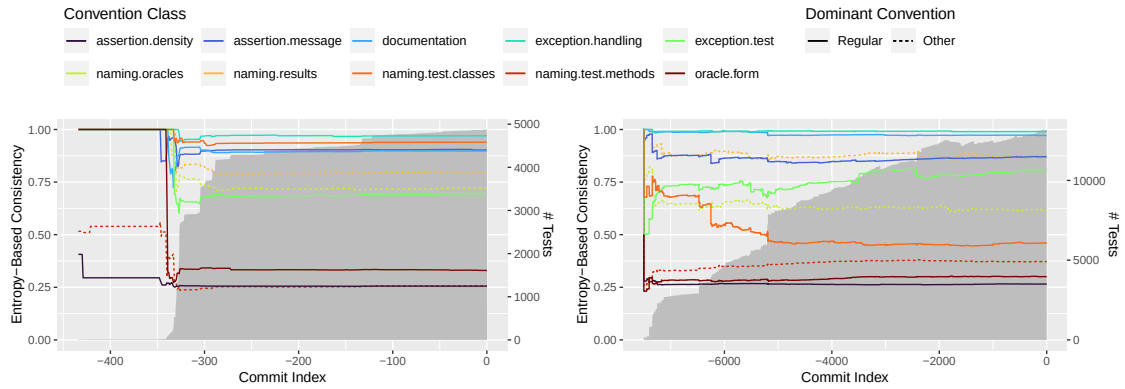
5.5 Analysis of Test Suite Consistency Change Events

We complemented our investigation of global trends by manually investigating the probable cause of consistency-changing events during the evolution of the projects. This investigation provided deeper insights into the factors that can affect consistency and how well our metrics capture this construct.

To select the events to investigate, we started by plotting the evolution of the entropy-based consistency of all convention classes for each project. Figure 8 shows two examples of such graphs.[27] Each line tracks the consistency of one convention class. The x axis shows the index of the commits as they are ordered on the default branch of the projects. Indices are shifted so that the latest commit has index 0. Henceforth, we refer to commits by their index to help locate the events on Figures 8 and 9. The shaded area shows the number of tests in the test suite to contextualize the evolution of consistency. The scale for the shaded area is shown on the alternative y axis on the right.

For some projects, such as Netty, the consistency remains almost constant after the initial fluctuations caused by a bulk addition of tests (Figure 8a). For other projects, such as Neo4j, there is more variation in consistency throughout

[27] Figures 8 and 9 show the evolution of only ten of the twenty studied projects. We include full-size graphs of all twenty projects in the online data artifact.



(a) Netty (b) Neo4j

Fig. 8. Evolution of entropy-based consistency. The shaded area shows the number of tests in the test suite, measured on the alternate y axis on the right.

the history of the project (Figure 8b). We sought interesting consistency-changing events in our target projects. For each selected event, we looked at the relevant commits, related pull requests and issue discussions, and distributions of convention instances, to assess whether the event was a consequence of deliberate actions from contributors. In the remainder of this section, we present each event and the results of our investigation.

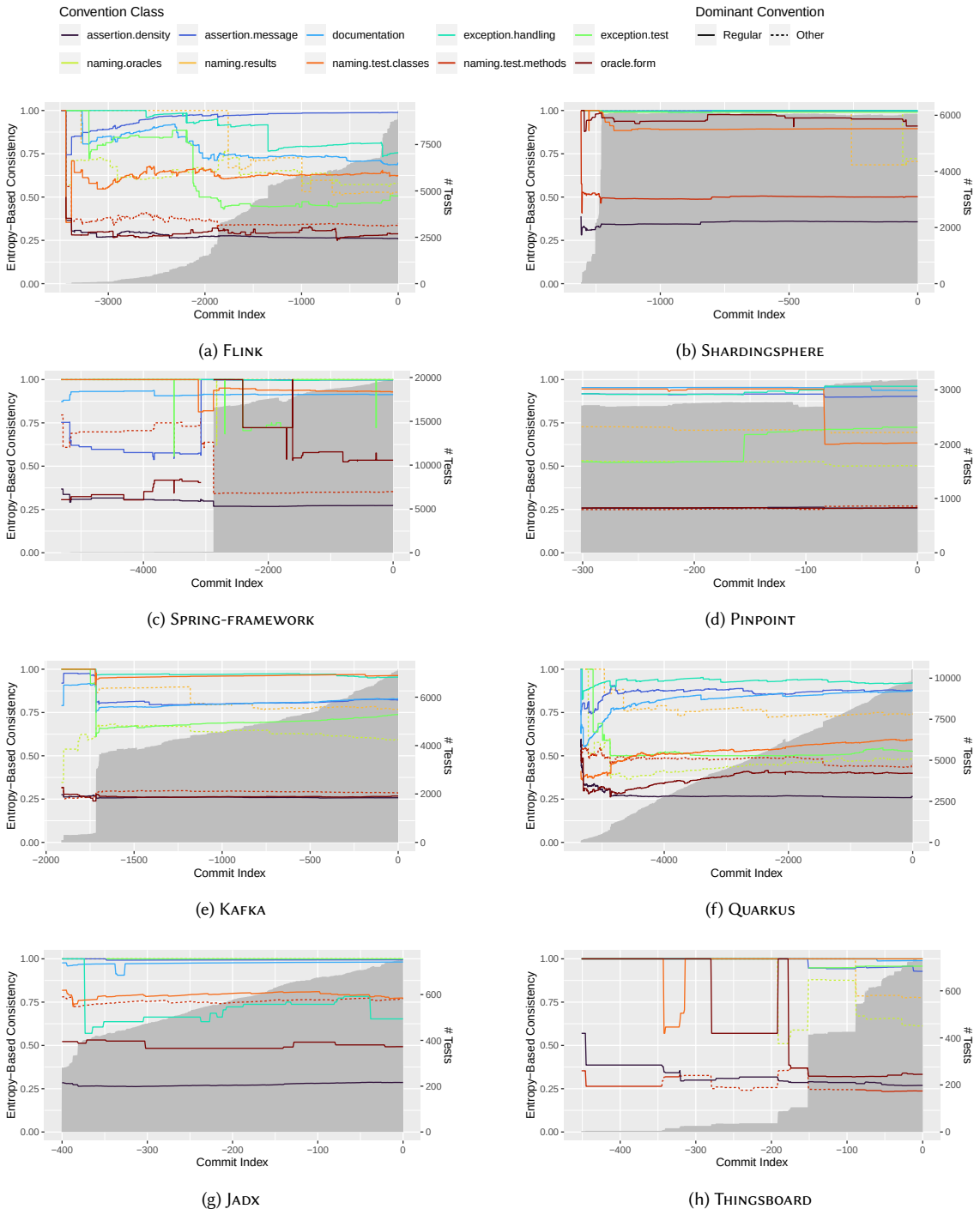
Short-term Fluctuation. We noticed some cases where the consistency dropped or increased considerably during a commit, then reset to its initial value a few commits later. For example, Flink shows such a spike at commit -721 for Testing Exceptions (Figure 9a), Shardingsphere has a spike at commit -482 for Specifying Oracles (Figure 9b), and Spring-framework has three spikes, at commit -2694 for Testing Exceptions and at commits -1707 and -276 for both Testing Exceptions and Specifying Oracles (Figure 9c).

In all cases, the initial change was contributed by a peripheral developer in a pull request, then corrected by a core developer.[28] In the case of Flink, the core developer suggested the correction in the pull requests, and the peripheral developer authored the changes themselves. In the case of Shardingsphere, the peripheral developer's pull request was merged with the inconsistencies, and the core developer opened a new pull request to "Revise #26845" (the original pull request), which was merged within ten minutes. In the case of Spring-framework, all three events were due to one of the core developers committing style revisions directly to the main branch. We observed that this core developer regularly contributes commits titled "Polishing".

These events show that core contributors spend effort monitoring and correcting the use of consistent conventions, even when those conventions are not documented. Our consistency scores could help reduce this effort by identifying sudden changes, both to allow core developers to target specific areas of the code that need correction and for peripheral developers to correct their contribution before submitting a pull request.

We also observed the practical limitations of the contributor classification approach during this investigation. For example, the original, peripheral developers were not systematically credited in the merged commits' authorship information. Conversely, reviews contributed by core developers are not retained in the commits' metadata. Thus,

[28] In this section, we use the terms core and peripheral developers to reflect the observable relationships between contributors. They may not align exactly with the criterion and its variations described in Section 5.4.



(a) Flink

(b) Shardingsphere

(c) Spring-framework

(d) Pinpoint

(e) Kafka

(f) Quarkus

(g) Jadx

(h) Thingsboard

Fig. 9. Evolution of entropy-based consistency for selected projects. Full-sized graphs are available in the online appendix.

accurately distinguishing contributions of core developers from peripheral ones requires a thorough analysis of multiple artifacts. We also noted that the use of testing libraries other than JUnit reduces the reliability of our consistency metrics. For example, many projects use the AssertJ library, which provides alternative assertion methods. These methods were not correctly identified as assertions during the data analysis, which led to the consistency scores ignoring legitimate instances of a convention class. This limitation stresses the importance of designing the conventions in a class to match the current and desired practices in a project.

Large Increase. The consistency of Testing Exceptions suddenly increases at commit -155 of Pinpoint (Figure 9d). This sudden and large increase contrasts with the otherwise relatively stable evolution of consistency and it does not coincide with the bulk insertion of tests seen at commit -83.

The commit message revealed that the intent of the contribution was precisely to refactor how exceptions are tested: "Cleanup Assertions.assertThrows". The commit replaced multiple instances of *Try Catch Idiom* with *AssertThrows*. For this event, our metric accurately reflects that the refactoring effort was successful in improving the global consistency of the test suite.

Large Decrease. In several projects, there are notable decreases of consistency that do not correlate with important variations in the number of tests. We selected three such decreases from Flink to investigate: in Testing Exceptions at commit -2123, in Naming Results at commit -1759, and in Handling Exceptions at commit -1347 (Figure 9a).

The decreases in Testing Exceptions and Handling Exceptions are attributable to the migration of test classes from version 4 to version 5 of JUnit. As we excluded JUnit 4 tests from our analysis, the migration had the effect to add new tests to our data set, which revealed inconsistencies that existed in the code but were not captured by the metrics. Nevertheless, the change in consistency reveals that that the developers did not adapt the code to the prevalent JUnit5 style during the migration, leading to an imbalance in conventions. In this case, the change in metric value indicates a missed opportunity to refactor testing strategies as part of a JUnit update. In contrast, the decrease of Naming Results consistency was caused by a developer introducing a new convention, *Actual Prefix*, whereas the test suite previously showed no evidence of a recognized convention. This event could indicate an effort to standardize the name of the result variable in code. A consistency metric can thus help notify other developers about an attempt to introduce a new convention.

Slow Constant Increase. Although it is not a punctual event, we observed a slow but constant increase in the consistency of Supplying Failure Message, Documenting Test Methods, and Testing Exceptions for Kafka, as well as a decrease in the (dominant) use of the catch-all *Other* convention of Naming Oracles (Figure 9e). We investigated whether this constant improvement was the result of a concerted effort to improve the quality of the test suite.

We could not find evidence of periodic refactorings to improve code style, such as those performed by the core developer of Spring-framework. However, we noticed that commit -1716, in which approximately half of the current test suite was added, was a large-scale migration from JUnit 4 to 5 in early 2021. Similarly to the previous event (*Large Decrease*), developers did not review the consistency of the imported tests during the migration, which generated initial inconsistencies. Since this migration, developers are more careful in applying a consistent style to new tests, which explains the increase in consistency. However, the initial inconsistencies were never resolved.

Increase with Periodic Drops. We noticed two cases where the consistency of a convention class would increase over time, but with multiple drops: for Documenting Test Methods in Quarkus (commits -3407 and -2092, Figure 9f) and for Naming Test Classes in Jadx (commits -275, -228, -178, -140, and -97, Figure 9g).

In the case of Quarkus, the drops happened during large pull requests authored and discussed between core developers (746 and 309 files changed, respectively). In both cases, the discussions of the pull requests suggest that the size of the pull requests and a desire to expedite merging them prevented a thorough review of code style. For example, the second pull request includes comments such as "This can be improved upon a lot, however it is sufficient for an initial attempt." and "Merging this as we need it in ASAP as to not conflict with other work". Together with the previous event (*Slow Constant Increase*), this event shows that even rare compromises in terms of consistency to prioritize other development concerns can have lasting consequences that counteract long-term effort to improve test suite quality.

In the case of Jadx, the periodic drops in consistency are due to two competing conventions being dominant in different folders of the project. With jadx-core/src/test/java/jadx/tests/, test classes in the integration/ subfolder mainly use the *Test Prefix* convention, whereas *Test Postfix* is more common in other subfolders. The two competing conventions decrease the global consistency score of the test suite.

Change of Dominant Convention. The dominant convention for Naming Test Methods in Thingsboard alternates several times between the catch-all *Other* and a regular convention, at commits -344, -340, -321, -175, -151, and -88 (Figure 9h).

We investigated whether these changes were conscious, e.g., due to conflicting ideas from different contributors. However, we found that they were mainly caused by a lack of control of the code style when developers contributed new tests. The low frequency of any convention, as well as the inconsistent use of helper methods which hid the method under test from our convention detection algorithm, explains the instability of the dominant convention.

RQ3: Which contextual factors can affect the consistency metrics and how? We found that our detection algorithm is limited by the use of helper methods and testing libraries, which can hide relevant instances of conventions in a class. A prevalent use of the catch-all convention in a class can artificially increase consistency scores, despite the code not following any fixed convention. However, these issues can be mitigated in practice through the design and implementation of convention detectors tailored to a project's technologies and cultural practices.

RQ4: What types of actionable insights are supported by an analysis of test suite consistency? The entropy-based consistency metric allowed us to successfully detect and quantify the impact of several developer actions, such as recurrent refactorings to improve the quality of tests suites. It also helped us identify events with a negative impact on test quality, such as compromises in test quality made to expedite the acceptance of large pull requests, which can introduce inconsistencies that remain several years in the project. Tracking the consistency of convention classes can thus help core developers monitor the quality of test suites, identify the location of inconsistencies, and externalize the knowledge about conventions used in a project.

5.6 Experimental Trade-offs and Threats to Validity

The design of our study required to accept trade-offs to present reliable and meaningful conclusions, at the cost of limitations and threats to validity related to some aspects of the investigation [47]. These trade-offs were the most desirable alternatives given our study goals and constraints.

Our study is limited by the precision of the detection algorithm to identify instances of a convention class. Accurately identifying all possible implementations of a concern, such as how to test an exception, is a complex challenge beyond the scope of this work. As a result, our observations may inaccurately represent the consistency aspect of test suite quality for the studied project, especially when they rely on testing libraries such as AssertJ. The decision to design generic convention classes, which are not tied to specific libraries, allowed us to apply the same systematic procedure

to study multiple open source projects. This decision suited our objective to assess the potential of consistency metrics to derive actionable insights to improve test quality.

Another limitation is due to our decision to implement the convention detection algorithm only for version 5 of JUnit. Many of the projects predate this version. Thus, instead of studying the natural growth of a test suite, our results are influenced by the adoption of the newer version: in some cases, this was done all at once, whereas the migration was gradual in other cases. As we did not measure the number of JUnit 3 or 4 tests, our analysis is oblivious to a potentially important part of the test suite. This decision was a trade-off between the accuracy of the detection algorithm and its coverage: the breaking changes in the assertion methods introduced in JUnit5, such as the position of the failure message argument, posed a threat to the reliability of the results. Given that JUnit5 was released over six years ago, we chose to prioritize the accuracy of the consistency metrics.

The decision to study twenty open source projects poses a threat to external validity. Although we sought to include large, collaborative software projects, we did not strive to gather a variety of projects along specific dimensions, such as the testing frameworks used, organizational management styles, or software domains. We also do not claim statistical representativeness to a larger population. Rather, we chose the number of projects under study, twenty, as a desirable compromise to generate a variety of consistency profiles while allowing a detailed manual investigation of each project.

Finally, inaccuracies in the author identity resolution pose a threat to the internal validity of the analysis of core and peripheral contributors' impact on consistency. Developers may use various {name, email address} combinations when authoring commits. Linking the various {name, email address} combinations a developer uses is necessary to classify their participation level correctly. Furthermore, we observed that commits added to the default branch of the projects did not always retain accurate authorship information. Author identity resolution, however, is a complex research problem in itself [13]. We attempted to mitigate this threat by performing email address and name unification using heuristics and acknowledge that our approach may inaccurately classify some developers' participation levels. As the objective of this analysis was not to derive definitive conclusions about the impact of peripheral contributors, we believe this threat is acceptable in the context of our study.

6 RELATED WORK

This research assumes a relation between code consistency and quality. This relation has been previously investigated outside the context of testing (Section 6.1). In the area of testing, prior work has surveyed developers to identify characteristics of high-quality tests (Section 6.2) and assess the state of existing unit tests (Section 6.3). Past work has also devised metrics to evaluate test quality (Section 6.4).

6.1 Code Consistency and Quality

Boogerd and Moonen contributed an early assessment of the correlation between violation of coding standard rules and detected faults for two industrial systems written in C [4, 5]. They investigated both the *temporal* and *spatial* co-occurrences of violations and faults, and discover a number of rules whose violations density correlates with fault density. Similarly, Smit et al. conducted a case study of the adherence to code conventions in four Java systems and provide descriptive statistics about the number of violations discovered, thereby documenting the extent to which many common conventions are not adhered to [52]. Takai et al., proposed 13 metrics based on code standard violations to assist in detecting latent faults in software system [57]. The different metrics capture aspects of the number and location of violations, but also the change in number of violations, and the software development effort associated with detected violations. This early work builds on the same assumption as we do, namely that adherence to conventions is

a marker of code quality. However, in addition to not targeting testing code, this early work considers only deviations between source code and a single, prescriptive coding standard. The main innovation of our work is that we consider consistency *independently of any given standard*, and target test code specifically.

Based on a survey of 55 computer science students and seven professional developers, Magalhães dos Santos et al. provided evidence that certain coding practices, such as avoiding multiple statements on the same line, are related to a higher perceived code readability [33]. This work helps strengthens the motivation for using coding conventions in general, but also leaves out the questions of general consistency we examine in this work. Closer to our work, Zu et al. study the difference in style between the code in pull requests on GitHub and the style in the code base, showing that "a [pull requests] that violates the current code style is likely to take more time to get closed" [66].

6.2 Characteristics of High-quality Tests

Multiple empirical studies surveyed developers to understand practitioners' perspectives on what constitutes high-quality unit tests. Kochhar et al. [27] 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 our research 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. [6] 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. [16] 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 [28, 35].

Tan et al. [58] 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 experts in the Swedish 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 research by highlighting that practitioners perceive readability and consistency as essential signals for test quality. In terms of methodology, our research 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.

6.3 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. [17] 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. [30] 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 [8] 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 article, we strive to advance towards this goal by exploring test convention consistency as a proxy for readability.

6.4 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 [37] 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 [9], which measures the percent of artificially injected defects a test suite can detect. Although mutation score has been observed to be more effective than code coverage [31], it is computationally expensive and, therefore, more seldom 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. [16] 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. [7]. 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. [10] 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 [18, 42, 43, 61] and investigate their relationship with various factors, such as error-proneness of source code [54]. One particular work by Bavota et al. [2] 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 article complements the above works by proposing a metric to capture another facet of test quality, namely consistency. Domain-specific test readability metrics, and other ensemble metrics, can leverage our proposed test convention consistency metrics as a feature within their models.

7 CONCLUSIONS

We investigated *test convention consistency* as one dimension of test quality and make four related contributions to support developers and other project stakeholders in improving and maintaining the quality of their tests.

Developing our initial contribution, the design of metrics for summarizing the consistency of a test suite, already surfaced some of the challenges that lay ahead. Not surprisingly, different metrics are sensitive to different macroscopic changes in test suite quality, and for this purpose we opted to propose a pair of metrics instead of a single one. Our technical analysis coupled with a systematic application of the metrics in practice showed how entropy-based consistency scores emphasize the negative impact of occasional deviations from a dominant convention whereas accuracy-based scores emphasize the most popular convention in convention classes that are less consistent. A second important challenge we faced early on is that consistency metric values are a function of the number and definition of conventions. Careful engineering effort must therefore be invested in the definition of conventions and their organization in convention classes, an endeavor we investigated with the development of a catalog of ten sample convention classes.

Designing the convention catalog allowed to collect, for the first time in one place, a wealth of knowledge on the various alternative of the test conventions classes. Previously, knowledge of test conventions was heavily fragmented, as style guides and articles in the grey literature typically only provide the one (or few) alternative preferred by the author. Our catalog can thus serve as a convenient menu of options to help project stakeholders consider convention alternatives. More importantly, however, creating the convention catalog forced us to consider numerous important design decisions related to the specification of individual conventions. For example, one central question when designing a convention class pertains to the target of the class, as different conventions can apply to different parts of the code (e.g., classes, methods, assertions). In the convention catalog, the *Design* and *Discussion* sections for each convention class provide insights on a large number of such design decisions that should benefit anyone undertaking the deployment of the approach in practice.

In a similar way, our development of the tool infrastructure to detect occurrences of conventions in production code surfaced numerous issues and concerns that can have a major practical impact. These includes what to define as a unit test, how to deal with multiple versions of JUnit, reliance on third-party matching libraries, and the impact of user-defined helper methods on the results. Hence, in addition to providing a blueprint for a tool to analyze test suite consistency, our experience developing TestComet and Teslo allows us to contribute numerous insights to inform project-specific adaptations and further improvements.

This research culminated in the use of TestComet to analyze over 80 000 test-related commits from the development history of 20 notable open-source Java projects. This empirical study provided several outcomes. First, it validated the design of the test convention classes and the implementation of TestComet by producing a data set of test convention occurrences we could manually check against the corresponding source code. Second, it provided descriptive statistics on the range of consistency values for ten different convention classes with our default conventions. Third, it enabled us to link observed changes in consistency values to specific events in the change history of our target systems, thus providing evidence of the construct validity of the metrics. Finally, the study allowed us to successfully detect and quantify the impact of several developer actions, such as recurrent refactorings to improve the quality of test suites or the deliberate acceptance of technical debt to integrate a pull request. We conclude that analyzing test suite consistency via static analysis shows promise as a practical approach to help improve test suite quality.

ACKNOWLEDGMENTS

This project is based on exploratory work by Alexa Hernandez, who contributed to a preliminary version of TestComet and Teslo and piloted the experiment. Elby MacKenzie contributed to an initial prototype of Teslo. We are grateful to the anonymous reviewers for their insightful suggestions, which motivated substantial improvements to this research. This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds de recherche du Québec—Nature et technologies (FRQNT).

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 281–293. https://doi.org/10.1145/2635868.2635883

[2] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. 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*. 470–481. https://doi.org/10.1109/SANER.2016.105

[4] Cathal Boogerd and Leon Moonen. 2008. Assessing the value of coding standards: An empirical study. In *Procedings of the 24th IEEE International Conference on Software Maintenance*. 277–286. https://doi.org/10.1109/ICSM.2008.4658076

[5] Cathal Boogerd and Leon Moonen. 2009. Evaluating the relation between coding standard violations and faults within and across software versions. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. 41–50. https://doi.org/10.1109/MSR.2009.5069479

[6] David Bowes, Tracy Hall, Jean Petric, Thomas Shippey, and Burak Turhan. 2017. How good are my tests?. In *Proceedings of the 8th IEEE/ACM Workshop on Emerging Trends in Software Metrics*. 9–14. https://doi.org/10.1109/WETSoM.2017.2

[7] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 107–118. https://doi.org/10.1145/2786805.2786838

[8] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering*. 201–211. https://doi.org/10.1109/ISSRE.2014.11

[9] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[10] Arie Van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. 2001. Refactoring test code. *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering* (2001), 92–95.

[11] Ramesh Fadatare. 2018. JUnit Framework Best practices. Personal blog. https://www.javaguides.net/2018/08/junit-framework-best-practices.html Verified 2024-05-24.

[12] Viktor Farcic. 2013. Test Driven Development (TDD): Best practices using java examples. Personal blog. https://technologyconversations.com/2013/12/24/test-driven-development-tdd-best-practices-using-java-examples-2/ Verified 2024-05-24.

[13] Tanner Fry, Tapajit Dey, Andrey Karnauch, and Audris Mockus. 2020. 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*. 518–522. https://doi.org/10.1145/3379597.3387500

[14] GitHub. 2024. About READMEs. GitHub Documentation Page. https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes Verified 2024-05-24.

[15] GitHub. 2024. Setting guidelines for repository contributors. GitHub Documentation Page. https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/setting-guidelines-for-repository-contributors Verified 2024-05-24.

[16] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. 2020. 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*. 336–347. https://doi.org/10.1109/ICSME46990.2020.00040

[17] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*. 348–351. https://doi.org/10.1145/3196321.3196363

[18] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*. 322–331. https://doi.org/10.1109/ICST.2013.45

[19] Lokesh Gupta. 2020. Unit testing best practices. HowToDoInJava article. https://howtodoinjava.com/best-practices/unit-testing-best-practices-junit-reference-guide/ Verified 2024-05-24.

[20] Philipp Hauer. 2022. Modern best practices for testing in Java. Personal blog. https://phauer.com/2019/modern-best-practices-testing-java/ Verified 2024-05-24.

[21] Oskar Jarczyk, Szymon Jaroszewicz, Adam Wierzbicki, Kamil Pawlak, and Michal Jankowski-Lorek. 2018. Surgical teams on GitHub: Modeling performance of GitHub project development processes. *Information and Software Technology* 100 (2018), 32–46. https://doi.org/10.1016/j.infsof.2018.

03.010

[22] JavaParser. [n. d.]. JavaParser Home Page. https://javaparser.org/ Verified 2024-05-24.

[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[24] Petri Kainulainen. 2018. Writing clean tests: Naming matters. Personal blog. https://www.petrikainulainen.net/programming/testing/writing-clean-tests-naming-matters/ Verified 2024-05-24.

[25] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2015. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21 (2015), 2035–2071. https://doi.org/10.1007/s10664-015-9393-5

[26] Vladimir Khorikov. 2019. You are naming your tests wrong! Personal blog. https://enterprisecraftsmanship.com/posts/you-naming-tests-wrong/ Verified 2024-05-24.

[27] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' views on good software testing practices. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*. 61–70. https://doi.org/10.1109/ICSE-SEIP.2019.00015

[28] Lasse Koskela. 2013. *Effective unit testing: A guide for Java developers*. Manning Publications.

[29] Ajitesh Kumar. 2021. 7 popular strategies: Unit test naming conventions. DZone article. https://dzone.com/articles/7-popular-unit-test-naming Verified 2024-05-24.

[30] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A. Kraft. 2016. Automatically documenting unit test cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. 341–352. https://doi.org/10.1109/ICST.2016.30

[31] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. 2009. 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*. 220–229. https://doi.org/10.1109/ICSTW.2009.30

[32] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2021. Mining fix patterns for FindBugs violations. *IEEE Transactions on Software Engineering* 47, 1 (2021), 165–188. https://doi.org/10.1109/TSE.2018.2884955

[33] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. 2018. Impacts of coding practices on readability. In *Proceedings of the 26th International Conference on Program Comprehension*. 277–2778. https://doi.org/10.1145/3196321.3196342

[34] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are static analysis violations really fixed? A closer look at realistic usage of SonarQube. In *Proceedings of the 27th International Conference on Program Comprehension*. 209–219. https://doi.org/10.1109/ICPC.2019.00040

[35] Robert C. Martin. 2009. *Clean code: A handbook of agile software craftmanship*. Prentice Hall.

[36] Gerard Meszaros. 2007. *Xunit test patterns: Refactoring test code*. Addison-Wesley.

[37] Joan C. Miller and Clifford J. Maloney. 1963. Systematic mistake analysis of digital computer programs. *Commun. ACM* 6, 2 (1963), 58–63. https://doi.org/10.1145/366246.366248

[38] Nam Ha Minh. 2019. JUnit tutorial for beginner with Eclipse. Personal blog. https://www.codejava.net/testing/junit-tutorial-for-beginner-with-eclipse Verified 2024-05-24.

[39] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 11, 3 (2002), 309–346. https://doi.org/10.1145/567793.567795

[40] Oracle. 1999. Code conventions for the Java programming language. https://www.oracle.com/java/technologies/javase/codeconventions-contents.html Verified 2024-05-24.

[41] Roy Osherove. 2013. *The art of unit testing* (2nd ed.). Manning Publications.

[42] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*. 311–322. https://doi.org/10.1109/ICSME.2018.00040

[43] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. 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*. 1650–1654. https://doi.org/10.1145/3368089.3417921

[44] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D. Newman. 2021. Using grammar patterns to interpret test method name evolution. In *Proceedings of the 29th International Conference on Program Comprehension*. 335–346. https://doi.org/10.1109/ICPC52881.2021.00039

[45] John Reese. 2022. Unit testing best practices with .NET Core and .NET Standard. Microsoft documentation article. https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices Verified 2024-05-24.

[46] Jon Reid. 2020. Unit test naming: The 3 most important parts. Personal blog. https://qualitycoding.org/unit-test-naming/ Verified 2024-05-24.

[47] Martin P. Robillard, Deeksha M. Arya, Neil A. Ernst, Jin L. C. Guo, Maxime Lamothe, Mathieu Nassif, Nicole Novielli, Alexander Serebrenik, Igor Steinmacher, and Klaas-Jan Stol. 2024. Communicating study design trade-offs in software engineering. *ACM Transactions on Software Engineering and Methodology* (2024), 9 pages. https://doi.org/10.1145/3649598

[48] Martin P. Robillard and Gail C. Murphy. 2003. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology* 12, 2 (2003), 191–221. https://doi.org/10.1145/941566.941569

[49] Martin P. Robillard, Mathieu Nassif, and Muhammad Sohail. 2024. Data from: Understanding test convention consistency as a dimension of test quality. Zenodo. https://doi.org/10.5281/zenodo.11267987

[50] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[51] Claude E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27, 3 (1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[52] Michael Smit, Barry Gergel, H. James Hoover, and Eleni Stroulia. 2011. Code convention adherence in evolving software. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*. 504–507. https://doi.org/10.1109/ICSM.2011.6080819

[53] Steve Ardalis Smith. 2011. Unit test naming convention. Personal blog. https://ardalis.com/unit-test-naming-convention/ Verified 2024-05-24.

[54] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*. 1–12. https://doi.org/10.1109/ICSME.2018.00010

[55] Stack Exchange Software Engineering Community. 2010. Is it OK to have multiple asserts in a single unit test? Stack Exchange Thread. https://softwareengineering.stackexchange.com/questions/7823 Verified 2024-05-24.

[56] Oleksandr Stefanovskyi. 2019. Unit test naming conventions. Medium article. https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea Verified 2024-05-24.

[57] Yasunari Takai, Takashi Kobayashi, and Kiyoshi Agusa. 2011. Software metrics based on coding standards violations. In *Proceedings of the Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*. 273–278. https://doi.org/10.1109/IWSM-MENSURA.2011.34

[58] He Tan, Vladimir Tarasov, and Anders Adlemo. 2018. Test case quality as perceived in Sweden. In *Proceedings of the 5th IEEE/ACM International Workshop on Requirements Engineering and Testing*. https://doi.org/10.1145/3195538.3195541

[59] Stack Overflow User. 2008. Unit test naming best practices [closed]. Stack Overflow Thread. https://stackoverflow.com/questions/155436 Verified 2024-05-24.

[60] Stack Overflow User. 2008. What are some popular naming conventions for Unit Tests? [closed]. Stack Overflow Thread. https://stackoverflow.com/questions/96297 Verified 2024-05-24.

[61] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. 2007. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering* 33, 12 (2007), 800–817. https://doi.org/10.1109/TSE.2007.70745

[62] Christian Vasquez. 2018. Introduction to unit testing with Java. DEV Community article. https://dev.to/chrisvasqm/introduction-to-unit-testing-with-java-2544 Verified 2024-05-24.

[63] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. 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*. 38–49. https://doi.org/10.1109/SANER.2018.8330195

[64] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. 2008. 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*. 220–229. https://doi.org/10.1109/ICST.2008.47

[65] Ahmed Zerouali and Tom Mens. 2017. 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*. 417–421. https://doi.org/10.1109/SANER.2017.7884645

[66] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. 2019. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. *Empirical Software Engineering* 24, 6 (2019), 3871–3903. https://doi.org/10.1007/s10664-019-09720-x