Measuring Software Testability Modulo Test Quality

Valerio Terragni
USI Università della Svizzera italiana
Switzerland
valerio.terragni@usi.ch

Pasquale Salza
University of Zurich
Switzerland
salza@ifi.uzh.ch

Mauro Pezzè
USI Università della Svizzera italiana
Schaffhausen Institute of Technology
Switzerland
mauro.pezze@usi.ch

ABSTRACT

Comprehending the degree to which software components support their own testing is important to accurately schedule testing activities, educate programmers and plan effective refactoring actions. Software testability estimates such a property by relating code characteristics with test effort.

The testability studies reported in the literature investigate the relation between class metrics and test effort, measured as the size and complexity of the associated test suites. They found that some class metrics have a moderate correlation with test-effort metrics. However, previous studies suffer from two main limitations: (i) their results hardly generalise because they investigated at most eight software projects; and (ii) they mostly ignore the quality of the tests. Considering the quality of the tests is important. Indeed, a class may have a low test effort because the associated tests are of poor quality and not because the class is easier to test.

In this paper, we propose an approach to measure testability that normalizes it with respect to the tests quality, which we quantified in terms of code coverage and mutation score. We present the results of a set of experiments on a dataset of 9,861 Java classes, belonging to 1,186 open source projects, with around 1.5 millions of lines of code overall. The results confirm that normalizing test effort with respect to test quality largely improves the correlation between class metrics and test effort. Better correlations would result in better prediction power, and thus better prediction of test effort.

CCS CONCEPTS

• **Software and its engineering → Designing software**; **Software testing and debugging**; *Software libraries and repositories*; *Software design tradeoffs*.

KEYWORDS

Software Testability, Test Effort, Software Metrics, Test Quality

ACM Reference Format:
Valerio Terragni, Pasquale Salza, and Mauro Pezzè. 2020. Measuring Software Testability Modulo Test Quality. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389273

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

1 INTRODUCTION

Software testing is an essential, labor intensive and time-consuming activity of the software life cycle. Making testing easier is important for many software companies, as it would reduce development costs and it would increase the number of detected faults.

It is well understood that the effort of testing software systems depends on the artifacts under test, that is, some software systems are easier to test than others [20, 28, 62]. Comprehending the relation between software artifacts and test effort is extremely important to control the cost of testing and improve the accuracy of test plans. *Software Testability* [28] captures the impact of software artifacts on testing by estimating *the degree to which a software system or component under test supports its own testing*[1].

In their recent comprehensive literature review of 208 papers on software testability, Garousi et al. [28] observe that measuring and predicting testability is the topic that received the most attention [7–9, 19, 20, 59]. The general idea is to measure (predict) the test effort of software systems from structural metrics of the software that are available before designing the test cases [59]. Early predicting the test effort can help developers to (i) early identify software components that require more test effort, on which developers have to focus to ensure software quality; (ii) plan testing activities and optimally allocate resources; and (iii) recognize refactoring opportunities to reduce the test effort.

Most studies on measuring and predicting testability focus on object oriented systems, and investigate the relation between class-level metrics, e.g., Chidamber and Kemerer (C&K) [21], and the cost of writing test cases (test effort) [7–9, 19, 20, 59]. These studies approximate the cost of writing test cases with the size and complexity of test suites, e.g., the number of tests and of assertions in the test class associated to the class under test. They provide some evidence of the existence of a correlation between class-level metrics and test effort, but suffer from two limitations: data sets of small size, and mostly ignore the quality of the test suites.

Small sample size. The previous studies involve at most eight software projects [28]. Such a small number of analyzed projects does not guarantee the generalizability of the results: specific development styles, frameworks and practices can influence the correlation results and produce different results for different projects [8].

Ignoring the test quality. The previous studies measure the test effort in terms of the size of the test classes, while mostly ignoring the quality of the tests. Lacking a quality assessment of the tests

[1] The literature proposes many definitions of software testability [28]. In this paper, we refer to the `IEEE 610.12-1990` and and `ISO/IEC 9126` standards [62] that define testability in a similar way: IEEE: *"the degree to which a system or a component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met"*. ISO : *"attributes of software that bear on the effort needed to validate the software product"*

leads to imprecise correlation results: classes with comparable test effort but different test quality should not have the same degree of testability. A class may have a low test effort because the associated tests are of poor quality, e.g., low line coverage, and not because the class is easier to test. Bruntink and Van Deursen's correlation study is the only work that acknowledges test quality when comparing test effort [20]. They ensured that the analysed software systems had test suites of similar quality (line coverage). However, their correlation study involves five software systems only [20].

In this paper, we propose a new approach to measure the testability of OO classes. *Our approach normalizes the test effort of a class with respect to the quality of its tests*, which we quantify with code coverage and mutation score. This enables correlation analyses and prediction models of an arbitrary large number of heterogeneous software systems implemented with different test quality criteria.

We empirically investigated our approach with 28 metrics to characterize the class properties, six metrics to measure the test effort, and three metrics to quantify the test quality. We analyzed 9,861 pairs of Java classes and corresponding JUnit test classes collected from 1,186 open source projects on GitHub. We computed the *Spearman's correlation coefficient* (ρ) [33] for all 168 pair-wise combinations of class and test-effort metrics, before and after the normalization with the test-quality metrics.

The results confirm that some class metrics correlates with test-effort metrics, and indicate that the normalization with test-quality metrics drastically improves the correlation (up to 74 %). Better correlation leads to better prediction power, and thus better prediction of test effort [59]. On the one hand, we used the test-quality metrics to normalize the test effort for the correlation analysis, allowing to fairly compare classes belonging to projects of different test qualities. On the other hand, the test-quality metrics can also be used as a target variable for prediction purposes. Indeed, if the purpose is to predict the test effort before writing the tests, a target value for test quality can be used in a preprocessing step to normalize the dataset used for training a prediction model. For instance, one might want to predict the test effort required to write tests having a target mutation score of 80 %. Thus, the data used to train the model would be normalized according to that value, building a model that predicts the test effort for the targeted mutation score of 80 %.

Moreover, the study indicates that (i) among the three test-quality metrics considered (line coverage, branch coverage and mutations score), normalizing by mutation score achieves the best correlation improvement, and (ii) the OO design properties that most influence testability are: size, complexity, coupling and cohesion.

This paper contributes to comprehend software testability by:

- presenting the by-far largest study on the correlation of class and test-effort metrics in terms of analyzed metrics, classes and projects;
- extending testability measurements by normalizing the test effort with respect to the quality of the test suites;
- showing that the proposed normalization improves the correlation between class metrics and test effort, leading to more accurate models for predicting test effort;
- giving important insights on software testability that confirm some of the findings of previous studies as well as uncover previously unknown correlations between OO design properties and test effort;
- publicly releasing our dataset for further studies[2].

The paper is organized as follows. Section 2 presents the objective of this study, and introduces the metrics considered. It also motivates and explains our normalization procedure. Section 3 presents the results of a series of research questions that validate the hypothesis that normalizing by test quality achieves better correlation than without normalization. Section 4 discusses the related work. Section 5 concludes the paper

.

2 OBJECTIVE AND METHODOLOGY

This paper investigates the relationships between object-oriented metrics of classes and the test effort of designing unit test cases for such classes. To achieve statistical significance and obtain general results for Java software systems, we conducted an experimental study on a large set of heterogeneous Java software systems of different size and category. Because different projects are likely to have different test quality criteria, in this paper we introduce a normalization procedure that homogenizes the values of test-effort metrics according to the values of test-quality metrics.

This section contextualizes the scope of the study (Section 2.1), presents the considered metrics of class, test effort and test quality (Section 2.2, Section 2.3, and Section 2.4, respectively), and introduces the new normalization procedure (Section 2.5).

2.1 Testability of Object-Oriented Programs

We target systems written in the **Object Oriented (OO)** programming paradigm [53], which is based on the concept of *"objects"* (instances of classes) that can contain both data (object fields) and code (methods). In particular, we consider systems written in the Java language.

Testing OO programs is often performed at three different levels [17]: unit, integration and system. *Unit testing* tests in isolation small portions of programs called units (e.g., methods or classes). The goal of unit testing is to isolate each part of the program and show that individual parts are correct. *Integration testing* tests the interaction of multiple units. *System testing* tests a complete and integrated software system.

When dealing with software testability, unit testing is the most useful testing level because one can apply testability analysis earlier in the development life-cycle [20]. Conversely, a testability analysis at system level requires a fully developed system. In line with the work presented in the literature [28], we study software testability at **unit level**, and more specifically at **class level**.

Considering class level testing has two practical advantages. First, we can leverage several OO metrics defined at class level [10, 21, 56]. Second, we can take advantage of popular naming conventions to identify the test class associated with a given class [20]. In fact, a common software development practice in OO programming

[2]The dataset and source code for the analysis we performed in this paper are shared at the address http://bit.ly/30uQoCk.

Table 1: Class metrics

Design Property	Name	Description	Reference
Size	Lines of Code (LOC)	Number of non-blank lines including comments and JavaDoc	–
	Number of Bytecode Instructions (NBI)	Number of bytecode instructions in the compiled `.class` file	–
	Lines of Comment (LOCCOM)	Number of lines of comment, excluding any end-of-line comments	–
	Number of Public Methods (NPM)	Number of methods in a class that are declared public	Goyal and Joshi [29]
	Number of STAtic Method (NSTAM)	Number of methods in a class that are declared static	–
	Number of Fields (NOF)	Number of fields (attributes) in the class	–
	Number of STAtic Fields (NSTAF)	Number of static fields (attributes) in the class	–
	Number of Method Calls (NMC)	Number of method invocations	–
	Number of Method Calls Internal (NMCI)	Number of method invocations of methods defined in the class	–
	Number of Method Calls External (NMCE)	Number of method invocations of methods defined in other classes	–
Complexity	Weighted Methods per Class (WMC)	Sum of the Cyclomatic Complexity [44] of all methods in the class	Chidamber and Kemerer [21]
	Average Method Complexity (AMC)	Average of the Cyclomatic Complexity [44] of all methods in the class	Tang, Kao and Chen [56]
	Response For a Class (RFC)	Number of methods that response to a message from the class itself	Chidamber and Kemerer [21]
Inheritance	Depth of Inheritance Tree (DIT)	Number of super classes	Chidamber and Kemerer [21]
	Number of Children (NOC)	Number of immediate sub-classes subordinated to a class in the class hierarchy	Chidamber and Kemerer [21]
	Measure of Functional Abstraction (MFA)	Ratio of the number of methods inherited by the class to the number of methods	Goyal and Joshi [29]
Coupling	Coupling Between Object classes (CBO)	Number of other classes that a class is coupled to	Chidamber and Kemerer [21]
	Inheritance Coupling (IC)	Number of parent classes to which a given class is coupled	Tang, Kao and Chen [56]
	Coupling Between Methods (CBM)	Number of redefined methods to which all the inherited methods are coupled	Tang, Kao and Chen [56]
	Afferent Coupling (Ca)	Measure of how many other classes use the specific class	Martin [42]
	Efferent Coupling (Ce)	Measure of how many other classes is used by the specific class	Martin [42]
Cohesion	Lack of Cohesion in Methods (LCOM)	Diff. between the number of method pairs without and with common variables	Chidamber and Kemerer [21]
	Lack of Cohesion Of Methods (LCOM3)	Revised version of LCOM	Henderson-Sellers [34]
	Cohesion Among Methods in class (CAM)	Represents the relatedness among methods of a class	Goyal and Joshi [29]
Encapsulation	Data Access Metrics (DAM)	Ratio of the number of private fields to the total number of fields	Goyal and Joshi [29]
	Number of PRIvate Fields (NPRIF)	Number of private fields (attributes) of the class	–
	Number of PRIvate Methods (NPRIM)	Number of private methods of the class	–
	Number of PROtected Methods (NPROM)	Number of protected methods of the class	–

languages, such as Java, is to create a dedicated class for each tested class following well-defined naming conventions [27, 48].

2.2 Class Metrics

Table 1 shows the 28 class metrics that we considered in this study. We decided to be as inclusive as possible when selecting the metrics, by avoiding to select only those metrics known to be correlated with testability [28]. We also wanted to see if unknown correlations arise with a large study. We considered well-known metrics that strongly correlate to the object oriented design: Chidamber and Kemerer (C&K) [21] and the Tang, Kao and Chen (TKC) [56] metrics. We enriched this already large set of metrics with additional metrics that may correlate with testability. Table 1 groups the 28 metrics based on the design property that each metric characterizes: *size*, *complexity*, *inheritance*, *coupling*, *cohesion*. and *encapsulation*.

Size. Size metrics includes standard ones such as Lines Of Code (LOC), metrics about the number of (static) methods and fields in the class (Number of Public Methods (NPM), Number of STAtic Method (NSTAM), Number of Fields (NOF), and Number of STAtic Fields (NSTAF)) and metrics about the number of internal and external method calls (Number of Method Calls (NMC), Number of Method Calls Internal (NMCI), Number of Method Calls External (NMCE)). In this paper we propose Number of Bytecode Instructions (NBI) as a new metric defined as the number of bytecode instructions in the compiled `.class` file. NBI can be more informative than the classic LOC metric, because single lines of code in Java can correspond to simple statements (e.g., variable assignment) or complex statements (e.g., lambda expressions), which correspond to few or many numbers of bytecode instructions, respectively. Thus, the NBI metrics distinguishes classes with similar number of LOCs but with different types of statements (complex and simple).

Complexity. Complexity metrics include Weighted Methods per Class (WMC) and Average Method Complexity (AMC). Both metrics depend on the *cyclomatic complexity* metric [44], which is its number of the linearly-independent paths of a method [44]. Because the cyclomatic complexity is defined at method level, WMC [21] and AMC [56] convert it to class-level by summing and averaging the cyclomatic complexities of all methods in the class, respectively.

Inheritance. Inheritance metrics capture the different aspects of inheritance of a class. Depth of Inheritance Tree (DIT) is the number of super-classes, Number of Children (NOC) is the number of immediate sub-classes in the class hierarchy, and Measure of Functional Abstraction (MFA) is the ratio of the number of methods inherited by the class to the total number of methods in the class.

Coupling. Coupling metrics characterize the degree of interdependence between classes and methods. Classes that have a high (outgoing) efferent coupling use other parts of the system, increasing the possible execution paths [51]. Low coupling is often a sign of a well-structured software system and a good design.

Cohesion. Cohesion metrics capture an important concept in OO programming. Cohesion describes the binding of the elements within one method and within one object class, respectively. Low cohesion means that the class does a great variety of actions.

Encapsulation. Encapsulation metrics capture the degree of encapsulation of the classes. For instance, the number of private methods

Table 2: Test effort metrics

Name	Description	Reference
TEST – Lines Of Code (T-LOC)	Number of non-blank lines including comments and JavaDoc of the test class	Bruntink and Van Deursen [20]
TEST – Number Of Tests (T-NOT)	Number of test cases in the test class (methods with the `@Test` annotation)	Bruntink and Van Deursen [20]
TEST – Number Of Assertions (T-NOA)	Number of test assertions in the test class (invocations to `org.junit.Assert`)	Bruntink and Van Deursen [20]
TEST – Number of Method Calls (T-NMC)	Number of method invocations in the test class	Toure et al. [57, 58]
TEST – Weighted Methods per Class (T-WMC)	Sum of the Cyclomatic Complexity [44] of all methods in the test class	Chidamber and Kemerer [21]
TEST – Average Method Complexity (T-AMC)	Average of the Cyclomatic Complexity [44] of all methods in the test class	Tang, Kao and Chen [56]

Table 3: Test quality metrics of T_C with respect to its associated class C

Name	Description	Reference
Line coverage (L)	Ratio of source code lines in C that are executed by at least one test in T_C	–
Branch coverage (B)	Ratio of branches in C that are executed by at least one test in T_C	–
Mutation score (M)	Mutation score of a test suite T_C with respect to a class C is the ratio of mutants that are *killed* by T_C	De Millo et al. [26]

and fields (NPRIM and NPRIF), the ratio of the number of private fields to the total number of fields (DAM).

2.3 Test-Effort Metrics

Test-effort metrics measure the effort of testing classes in terms of the size and complexity of the associated test cases. We refer to test classes written in JUnit, the most popular testing framework for Java [27, 48]. Let $\boldsymbol{T_C}$ denote the JUnit **test class** of a class $\boldsymbol{C}$. Each test method in T_C includes zero or more assertion oracles (boolean conditions) that predicate on the behavior of the class under test, and whose runtime values determine the pass or fail status of the test case. A test class may declare additional methods and inner classes to support the test executions, and such test code is called *test scaffolding*. Examples of scaffolding methods are those annotated with `@Before` and `@After`.

The effort of testing a class C is best quantified using the man-hours required to design and implement T_C [43]. However, collecting such a information for few software projects is difficult [43], and becomes unrealistic for many software projects. As such, researchers often approximate the test effort with the size and complexity of the test class (*test-effort metrics*) [7, 8, 19, 20, 59]. Relying on the assumption that the size and complexity of a test class reflect the effort for writing and designing it.

Table 2 shows the six test-effort metrics that we use to characterized test effort. The first four metrics TEST - Lines Of Code (T-LOC), TEST – Number Of Tests (T-NOT), TEST – Number Of Assertions (T-NOA), and TEST – Number of Method Calls (T-NMC) measure the size of the test class under different perspectives. T-LOC considers the size of the entire test class, and thus it includes scaffolding methods and inner classes. T-LOC also considers comment lines, since adding comments to the test code contributes to the cost of designing test cases [7]. T-NOT indicates the number of test cases regardless of their size. T-NOA captures the amount of assertions contained in the test class, which might not be the same as T-NOT, because a test may have multiple assertions. T-NMC (called TINVOKE in Toure et al.'s paper [59]) measures the number of method calls in the test class, which is a proxy for the degree of dependency of the test cases [59]. According to Toure et al., this metric is important to characterize the test effort of classes [58]. Intuitively, a test class with few dependencies is easier to execute than a test class with many dependences, because a test class with few dependencies can invoke the methods under test directly [59]. TEST – Weighted Methods per Class (T-WMC) and TEST – Average Method Complexity (T-AMC) measure the cyclomatic complexity [44] of the test class, which well quantifies the test effort. A test class may contain test or scaffolding methods with many linearly independent execution paths. We can expect that the higher the complexity of the test class the higher the effort required for writing and designing it.

2.4 Test-Quality Metrics

In this paper, we propose the use of test-quality metrics to normalize test-effort metrics. Table 3 shows the three test-quality metrics that we considered: *line coverage*, *branch coverage* and *mutation score*. These metrics are commonly used to approximate the quality of a test suite defined as the ability to reveal faults [62].

Coverage analysis [49] executes the test class T_C on an instrumented version of the class under test C to collect coverage information, and computes the percentage of structural code that T_C executes. Intuitively, executing the faulty statements is a necessary (but not sufficient) condition to expose software faults. As such, test coverage is a test quality metric because the higher the coverage of T_C, the higher the chance that T_C executes faulty statements (if they exist) [36]. In our experiments we compute statement and branch coverage, the two most popular code coverage metrics [36].

Even if a test suite covers faulty statements, it may not reveal the faults. An effective test suite needs test oracles (test assertions) that are able to distinguish correct from incorrect program behaviors. DeMillo et al. introduced **mutation analysis** in the late seventies [26] offering an alternative approach to evaluate the effectiveness of a test suite. It seeds artificial faults in the class under test, producing faulty versions (called mutants). Each of these mutant contains a single seeded fault. Then, mutation analysis executes the test suite on each mutant and counts how many mutants the test suite "kills", i.e., at least one test fails because of the seeded fault. It then computes the mutation score of the test suite as the percentage of mutants killed when executing the test suite. The mutation score evaluates not only the ability of tests to cover the seeded faults but also the ability of test oracles to expose such faults.

The values of the test-effort metrics range from zero to one. For example, a line coverage of 0.5, means that a test suite T_C covered half (50 %) of the source code lines in the class under test C.

2.5 Normalization

In this paper we investigate the use of test-quality metrics as normalization factors when measuring test effort. Current testability approaches measure the test effort by referring only on the size and complexity of the test cases (see Table 2) [19, 20, 55, 59], mostly ignoring the adequacy of the tests (test quality). Focusing only on the size and complexity of test suites without considering their quality may be misleading. A small test suite may be due to the easiness of writing a high quality test suite, indeed, and thus indicate high code testability, but may also be due to a bad quality test suite, and be completely unrelated to the code testability.

Ignoring the quality of the test suite produces imprecise correlation results, and thus imprecise prediction models if test suites of different quality are considered. For instance, let us consider two classes C_1 and C_2, and the corresponding test classes T_{C1} and T_{C2}. Let us assume that C_1 has 1,000 lines of code (LOC = 1,000), T_{C1} has ten test cases (T-NOT = 10), and T_{C1} covers 10 % of the source code lines of C_1 (line coverage = 0.1). Let us also assume that C_2 has LOC = 50, T_{C2} has T-NOT = 30, and line coverage is 90 %. Approaches based solely on the size and complexity of the test cases would indicate higher testability (i.e., lower test effort) for C_1, which requires only 10 test cases for 1,000 lines of code, than C_2, which requires 30 test cases for 50 lines of code. However, the small size of T_{C1} comes with a very low coverage, which indicates a very low quality of the test suite, while the relatively large size of T_{C2} comes with a very high coverage, which indicates a very high quality of the test suite. The issue is that T_{C1} and T_{C2} have a considerably different test quality (10 % and 90 % line coverage), and thus it is meaningless to compare the number of tests of T_{C1} and T_{C2} for studying their correlations with class metrics.

Bruntink and Van Deursen's study is the only work that acknowledges test quality when measuring test effort [20]. Their study suggests the importance of the quality of test suites when investigating software testability, but removes the issue by selecting subjects with test suites that achieves the same code coverage. This approach is possible when dealing with as small and homogeneous set of subjects, but becomes impractical when dealing with many heterogeneous subjects implemented with different test adequacy criteria (as in our case).

We address this issue by normalizing the test-effort with the test-quality. In our approach, we compute both test-effort and test-quality metrics for each test class T_C, and then "normalize" the test-effort of T_C with the actual test-quality of T_C.

Our procedure normalizes the values of each test-effort metrics for all the analyzed systems proportionally to a fixed target test-quality. Our normalization is grounded on the intuition that test effort grows with an increased test quality. We normalize test effort over test quality as:

$$\frac{\textit{normalized test-effort value}}{\textit{target test-quality value}} = \frac{\textit{actual test-effort value}}{\textit{actual test-quality value}}$$

The *target test-quality value* is a fixed decimal number between zero (excluded) and one. Intuitively, both code coverage and mutation score is a decimal number within such a range. In our experiment, for simplicity we consider *target test-quality value* to be 1 (but we could have chosen any possible value in the range (0;1]).

For the example discussed above, with a *target test-quality value* of 1, the normalized value of T-NOT with respect to line coverage is $\frac{10}{0.10} = 100$ for T_{C1} and $\frac{30}{0.90} = 33.33$ for T_{C2}. After the normalization, 1,000 LOCs of C_1 relates with T-NOT = 100 and 50 LOCs of C_2 relates with T-NOT = 33.33, resulting in comparable values.

In the next section, we present our experiments to investigate if such normalization procedure improves the correlation of class metrics with respect to test effort.

3 EXPERIMENTAL RESULTS

This section describes the results of a set of experiments that evaluate our proposed approach for measuring software testability. We addressed two research questions:

RQ1 *What is the correlation between class and test-effort metrics?*

RQ2 *Does the normalization with test-quality metrics increase correlation?*

We released the dataset and the source code of the analysis that we performed in this paper at the address http://bit.ly/30uQoCk.

3.1 Data Collection

We selected 1,186 Java open-source projects from GitHub, the most popular platform for code hosting. We excluded low quality and toy projects, by considering only those Java projects with at least 50 stars and at least one fork. We queried GitHub to obtain the list of public repositories that match our criteria. We implemented an automated script that clones the latest version of the master branch for each repository in the list, and selects all the repositories that (i) contain at least one JUnit test class, which we identify from an import declaration with package `org.junit`, and (ii) use either Gradle or Maven as build automation systems (which we identify from the presence of the file `build.gradle` or `pom.xml`). (iii) build successfully with no failing tests. This is because mutation analysis needs a "green" test suite [26]. We require either Gradle or Maven because we need build automation systems to automatically build the projects, collect and resolve the runtime dependencies and run test cases. Indeed, we need compiled code to compute most class and test-effort metrics (see Table 1 and Table 2), and we need to execute the test cases to compute the test quality metrics (see Table 3). Gradle and Maven are among the most popular build automation systems for Java. We found 1,186 GitHub projects that satisfy these three conditions.

Extracting pairs of class and test class. We automatically explored the content of each of the 1,186 projects to extract the pairs $\langle C, T_C \rangle$, where C is a Java class and T_C is the JUnit test class associated to C. This is in line with the typical usage of JUnit for unit testing [59], test a class C by means of a dedicated class T_C. Following previous work that study the correlation between class and test effort metrics [19, 20, 30], we identified such pairs by relying on the JUnit naming conventions for test classes [27, 48]. The name of the test class T_C should be the name of the associated class C plus the word "Test" or "TestCase" as prefix or suffix [48]. For example, if a

Table 4: Descriptive statistics of class metrics

Metric	Mean	SD	Min	Q1	Q2	Q3	Max	Metric	Mean	SD	Min	Q1	Q2	Q3	Max
LOC	161.68	226.61	8.00	60.00	99.00	177.00	6,758.00	NOC	0.03	0.33	0.00	0.00	0.00	0.00	19.00
NBI	368.64	917.82	5.00	84.00	183.00	397.00	60,250.00	MFA	0.21	0.33	0.00	0.00	0.00	0.48	1.00
LOCCOM	149.97	218.04	0.00	52.00	91.00	170.00	6,267.00	CBO	6.72	8.17	0.00	2.00	4.00	8.00	156.00
NPM	7.98	11.95	0.00	2.00	5.00	9.00	247.00	IC	0.33	0.59	0.00	0.00	0.00	1.00	5.00
NSTAM	2.33	6.35	0.00	0.00	1.00	2.00	141.00	CBM	0.65	2.04	0.00	0.00	0.00	1.00	48.00
NOF	4.50	31.73	0.00	1.00	2.00	4.00	2,083.00	CA	0.71	4.00	0.00	0.00	0.00	1.00	151.00
NSTAF	2.21	31.45	0.00	0.00	0.00	1.00	2,083.00	CE	6.04	7.22	0.00	2.00	4.00	8.00	108.00
NMC	46.52	111.41	1.00	10.00	22.00	50.00	6,638.00	LCOM	104.01	755.86	0.00	1.00	6.00	30.00	31,688.00
NMCI	16.10	46.40	0.00	2.00	6.00	16.00	2,533.00	LCOM3	0.90	0.65	0.00	0.50	0.75	1.10	2.00
NMCE	30.42	79.16	0.00	5.00	14.00	34.00	4,165.00	CAM	0.43	0.19	0.02	0.29	0.40	0.56	1.00
WMC	10.50	13.56	1.00	4.00	6.00	12.00	2,560.00	DAM	0.68	0.44	0.00	0.00	1.00	1.00	1.00
AMC	23.34	43.34	0.11	9.67	16.29	27.00	1,734.50	NPRIF	2.74	4.32	0.00	0.00	2.00	3.00	117.00
RFC	29.73	30.36	2.00	12.00	20.00	37.00	520.00	NPRIM	1.41	3.56	0.00	0.00	0.00	1.00	95.00
DIT	1.49	0.84	1.00	1.00	1.00	2.00	8.00	NPROM	0.43	1.75	0.00	0.00	0.00	0.00	82.00

Table 5: Descriptive statistics of test-effort metrics

Metric	Mean	SD	Min	Q1	Q2	Q3	Max
T-LOC	112.83	126.18	10.00	50.00	77.00	127.00	3,013.00
T-NOT	5.01	6.98	1.00	1.00	3.00	6.00	191.00
T-NOA	9.26	23.56	0.00	0.00	3.00	9.00	784.00
T-NMC	69.91	116.68	1.00	17.00	36.00	76.00	2,377.00
T-WMC	7.27	8.90	2.00	3.00	5.00	8.00	196.00
T-AMC	34.46	43.69	1.06	15.00	24.25	40.00	1,605.69

Table 6: Descriptive statistics of test-quality metrics

Metric	Mean	SD	Min	Q1	Q2	Q3	Max
Line coverage (L)	0.78	0.26	0.00	0.67	0.88	1.00	1.00
Branch coverage (B)	0.67	0.31	0.00	0.50	0.75	1.00	1.00
Mutation score (M)	0.65	0.32	0.00	0.40	0.71	1.00	1.00

class name is `Connector`, the name of the JUnit test class should be either `ConnectorTest` or `ConnectorTestCase`. As shown by previous studies [19, 20, 30], Java developers often respect such a convention. As a result, this approach precisely identifies the pairs $\langle C, T_C \rangle$ of class and associated test class [19, 20, 30].

Collecting class metrics. For each of the analyzed Java classes, we collected the class metrics with a static analyzer that we implemented on top of `CKJM-extended` [2] by Jureczko and Spinellis [38], currently the most comprehensive open-source tool to compute OO metrics for Java [38]. `CKJM-extended` computes 18 [4] of the class metrics in Table 1. We implemented additional static analyzers to compute the remaining 11 class metrics. The static analyzers compute the 28 class metrics (Table 1) taking in input the source code of C and the JARs outputted by the build automation system, which contain the compiled classes of C's project and its runtime dependencies. This is because the computation of some metrics require the source code of the class C, while others require the compiled binary code of the class C.

Collecting test-effort metrics. Table 2 presents the test-effort metrics from T_C that we collected with our static analyzer that already computes LOC, WMC, AMC and NMC. We implemented additional static analyzers for computing the remaining test-effort metrics, i.e., T-NOT, and T-NOA. The static analyzer computes the six test-effort metrics by taking in input the source code of T_C and the JARs outputted by the the build automation system.

Collecting test-quality metrics. We collected the test-quality metrics (Table 3) relying on the latest versions of JaCoCo [3] for code coverage and on PIT [5] for the mutation score. We built an automated script that modifies the `build.gradle` and `pom.xml` build configuration files of each project by adding JaCoCo and PIT dependencies. The scripts automatically invokes JaCoCo and PIT (via Gradle or Maven), which execute each test class T_C individually to compute its line and branch coverage and mutation score.

Dataset. We ran the tools and scripts on all the 1,186 project cloned from GitHub. We aggregated the results by automatically parse the report files of the static analyzer, JaCoCo and PIT. In total, we computed all the metrics for 9,861 pairs $\langle C, T_C \rangle$ of class C and associated test class T_C. We counted an average of 8.31 pairs $\langle C, T_C \rangle$ per project. The 9,861 C and T_C classes have a cumulative LOC of 1,594,309 and 1,112,652, respectively. T_C classes have 49,413 test cases overall and 5.01 on average.

Table 4, Table 5 and Table 6 show the descriptive statistics of the values of the class, test-effort, and test-quality metrics in our dataset, respectively. The tables show for each metric the average (column "Mean"), standard deviation (column "SD"), minimum value (column "Min"), first quartile (column "Q1"), second quartile (column "Q2"), third quartile (column "Q3") and maximum value (column "Max").

Our dataset contains classes with a wide range of structural and OO design properties (Table 4) and test classes with different size and complexity (Table 5). Table 6 indicates that the 9,861 test classes have a considerable amount of variation of the three test-quality metrics (Standard Deviation (SD) ~ 0.30). This confirms our hypothesis that test quality criteria vary largely among open-source projects, and motivates the need of our normalization adjustment, as discussed in Section 2.5.

The median (Q2) and mean of the test-quality metrics are relatively high, which indicate that the projects are well-tested. Line coverage, branch coverage and mutation score have a median of

0.88, 0.75 and 0.71, respectively. This may be related to the fact that we only selected popular, projects with at least 50 stars and at least one fork. Few test classes have zero coverage and mutations score (Column "Min" of Table 6). In such cases, the JUnit naming convention was not effective in identifying the pairs of C and T_C. We excluded these pairs from our analysis.

3.2 RQ1 – Correlation Study

To answer RQ1, we computed the "Spearman's correlation coefficient" (ρ) [33] for all 168 (28×6) pair-wise combinations of class and test-effort metrics. *Spearman*'s coefficient is a popular non-parametric measure of correlation used by related studies [20, 59].

We opted for a non-parametric statistical measure because none of the metric's observations (see Table 4, Table 5 and Table 6) follow a normal ("Gaussian") distribution. And thus, parametric measures of correlation (e.g., "Pearson"), cannot be used [33]. We checked for normality with the "D'Agostino's K^2" test [25]. Other normality tests (e.g., "Shapiro-Wilk Test"), are inadequate because each distribution has more than 5,000 data points [52]. The K^2 test calculates the *kurtosis* and *skewness* to determine if a data distribution departs from the normal distribution [25]. For each of the metrics's value distributions, the normality test leads to $p\text{-}value \leq \alpha$ ($\alpha = 0.05$), and thus we reject the hypothesis that are normal distributions.

Spearman's coefficient (ρ) quantifies the degree to which two variables are associated with a monotonic function, i.e., an increasing or decreasing relationship [23]. The coefficient ρ ranges from -1 to $+1$. A positive ρ means that both variables increase together. Instead, a negative ρ means that both variables decrease together. A ρ close to zero means that the two variables have no correlation.

Figure 1 shows the heatmap of the *Spearman*'s coefficients for each of the 168 pair-wise combinations of class and test-effort metrics. For this research question we are not interested in distinguishing positive and negative correlations, and thus Fig. 1 reports absolute values $|\rho|$. The colors range from green (max correlation $|\rho| = 1.0$) to red (min correlation $|\rho| = 0.0$).

We interpret $|\rho|$ as **weak ($\leq$ 0.3)**, **moderate (0.3 – 0.5)**, or **strong ($\geq$ 0.5)**, following the widely accepted classification of Cohen [22]. The column "Without Normalization (N)" in Table 7 shows the number of class metrics that have weak, moderate and strong correlations for each test-effort metrics.

The *p-value* (probability value) computed for each moderate and strong correlations is always less than 0.0001, and thus we can reject the null hypothesis that the metrics are uncorrelated. This result confirms those of previous studies [19, 20, 59]. Some class and test-effort metrics have moderate correlations (39 in our dataset).

3.3 RQ2 – Normalization Effect on Correlation

To answer RQ2, we normalized each value of the test-effort metrics with the corresponding value of a test-quality metric, using the formula described in Section 2.5 using 1 as target test quality value. For example, when considering line coverage, a target test-quality value of 1 means normalizing by 100 % line coverage. Because we considered three test-quality metrics (see Table 3), we obtained three variants of our original dataset. Each variant consider a distinct test-quality metric: **L**ine coverage (**L**), **B**ranch coverage (**B**) and **M**utation score (**M**). For each variant, we recomputed the Spearman's coefficient ($|\rho|$) for the 168 pair-wise combinations of class and (normalized) test-effort metrics.

Figure 2, Fig. 3 and Fig. 4 show the heatmaps of $|\rho|$ after each normalization. Compared with Fig. 1, the values of $|\rho|$ coefficients drastically increase. Table 7 shows the number of weak, moderate and strong correlations after each normalization. Normalizing by Line of coverage (L) decreases the number of weak correlations from 128 to 98 and increases the number of moderate and strong correlations from 39 to 52 and from 1 to 18, respectively. Normalizing by Branch of coverage (B) achieves similar results. Normalizing by Mutations score (M) leads to the best correlation improvement. The number of weak correlations decreases from 128 to 83, while the number of moderate and strong correlations increases from 39 to 51 and from 1 to 34, respectively.

To better quantify the correlation improvement we compared the $|\rho|$ coefficients before and after normalization. We started by removing all combinations of class and test-effort metrics that characterize negligible or not-existing correlations. Removing them is important because their $|\rho|$ values are "statistical fluke" [22], which is a result that you get simply by chance, not because there is a correlation [22]. Recognize negligible or not-existing correlations is nontrivial because several $|\rho|$ values that are close to zero (weak) before normalization become higher (moderate) after (see Table 7).

As such, we removed *all combinations* of class and test-effort metrics that correspond to $|\rho| < 0.1$ *both before and after normalization*, which likely represents statistical flukes [22]. Line coverage normalization has 51 of such combinations, Branch normalization 50 and Mutation normalization 42. The p-value for such combinations confirms that the little visible correlation is a *statistical fluke*. Indeed, 91 (63.63 %) of the 143 removed combinations have a p-value > 0.0001, whereas the p-values of all retained combinations is < 0.0001. Therefore, we can reject the null hypothesis that the retained combinations are uncorrelated.

For each type of normalization, we computed $\Delta|\rho|$ as the difference between the $|\rho|$ values after and before normalization. For example, if we consider the combination LOC and T-NOT, $|\rho|$ before normalization ($|\rho|_N$) is 0.35 and after normalization by mutation score ($|\rho|_M$) is 0.58. Then, $\Delta|\rho| = |\rho|_M - |\rho|_N = 0.58 - 0.35 = 0.23$. The Line normalization retains 117 combinations. Their $\Delta|\rho|$ is 0.09 on average (max 0.17 and min -0.02). Only in four cases the Spearman's correlation decreases after normalization (i.e., $\Delta|\rho| = |\rho|_L - |\rho|_N < 0$). Instead, the Branch normalization retains 118 combinations. Their $\Delta|\rho|$ is 0.08 on average (max 0.18 and min -0.04). Only in nine cases the Spearman's correlation decreases after normalization. Finally, the Mutation normalization retains 126 combinations. Their $\Delta|\rho|$ is 0.13 on average (max 0.26 and min -0.008). Only for the combination CA and T-NOA the correlation decreases after normalization (i.e., $\Delta|\rho| = |\rho|_M - |\rho|_N = -0.008$). Among the three normalizations, the Mutation normalization leads to the best correlation improvement. This could be due to mutation score is better than code coverage in evaluating a test suite [26]. It evaluates both the ability of tests to cover the faulty statements and the ability of test oracles to detect the faults.

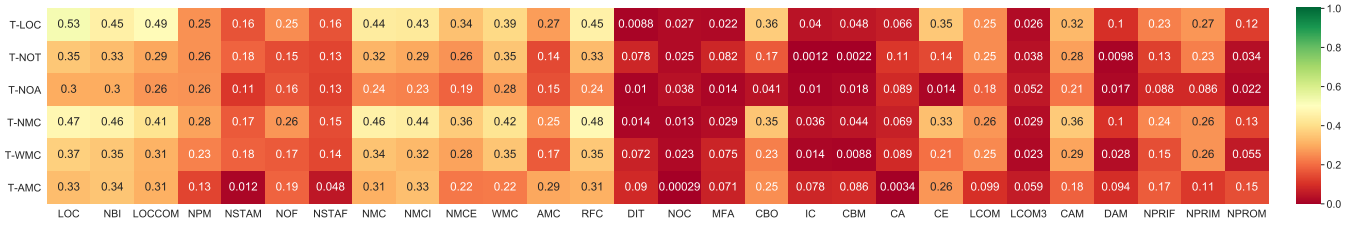

Figure 1: Spearman rank correlation coefficient (absolute values) – without Normalization (N)

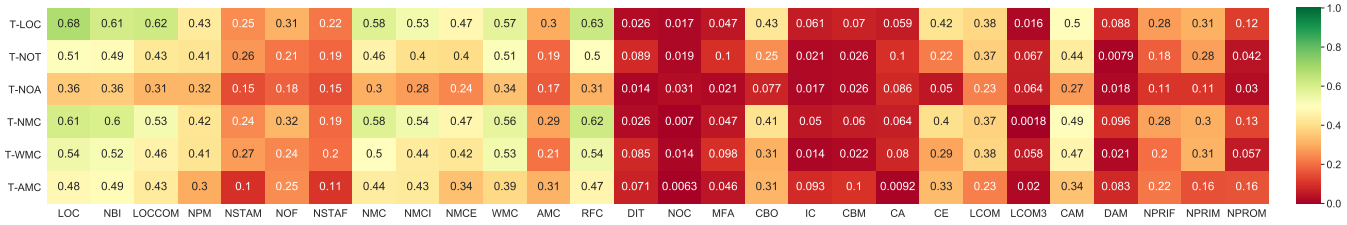

Figure 2: Spearman rank correlation coefficient (absolute values) – test effort metrics are normalized by Line coverage (L)

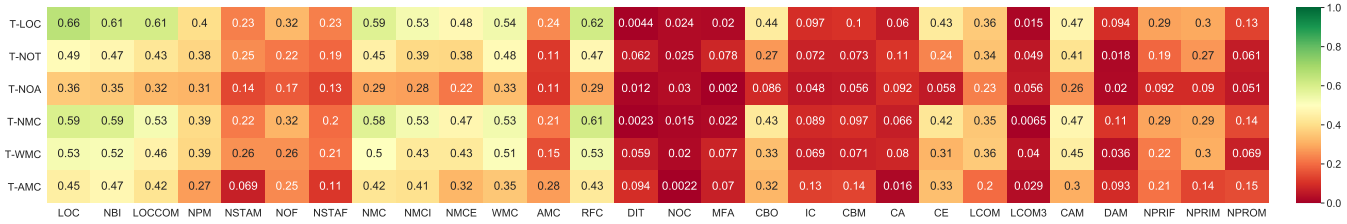

Figure 3: Spearman rank correlation coefficient (absolute values) – test effort metrics are normalized by Branch coverage (B)

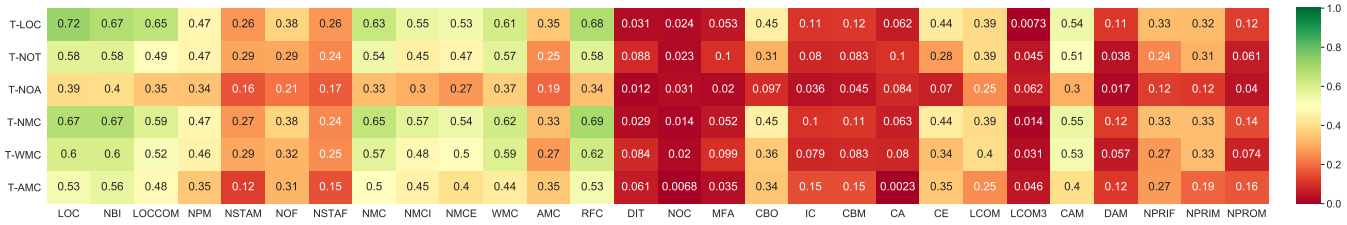

Figure 4: Spearman rank correlation coefficient (absolute values) – test effort metrics are normalized by Mutation score (M)

3.4 Discussion of the Results

Table 8 shows the Spearman's coefficients of the 12 class metrics that most correlate with test effort, i.e., they have at least one $|\rho|$ value ≥ 0.4 (either before or after normalization). We chose 0.4 as the threshold because it includes strong correlations and the highest half of moderate correlations [22]. Table 8 highlights in bold the highest $|\rho|$ value for each column. All selected class metrics have a positive correlation, except CAM that has a negative one ($\rho < 0$).

These 12 class metrics belong to four OO design properties: size, complexity, coupling and cohesion (see Column "Design Property" Table 1). More specifically, seven metrics characterize the size of the class (LOC, LOCCOM, NBI, NMC, NMCI, NMCE, NPM), two the complexity (RFC and WMC), two the coupling (CBO and CE), and one the cohesion (CAM). These results confirm some of the findings of related studies [7, 8, 20, 59] as well as report new correlations.

We now discuss the similarities and differences of our findings with the ones of two representative studies: one by Bruntink and Van Deursen [20] and one by Toure et al. [59]. We chose these studies from the literature because they analyze the highest number of projects and class/test-effort metrics. Referring to the published results, we consider a class metrics to be highly correlated to test effort if (i) the $|\rho|$ value ≥ 0.4 with respect to any test-effort metric; (ii) the correlation results are statistical significant.

Bruntink and Van Deursen [20] studied five open-source Java programs to compute the Spearman's correlation between nine Chidamber and Kemerer (C&K) metrics (DIT, NMC, LCOM, LOC, NOC,

Table 7: Counting the Spearman's coeff. absolute values $|\rho|$ as weak ($|\rho| \leq 0.3$), moderate ($0.3 < |\rho| < 0.5$), or strong ($|\rho| \geq 0.5$)

Test effort	Without Normalization (N)			Normalized by Line coverage (L)			Normalized by Branch coverage (B)			Normalized by Mutation score (M)		
	Weak	Moderate	Strong	Weak	Moderate	Strong	Weak	Moderate	Strong	Weak	Moderate	Strong
T-LOC	17	10	1	15	6	7	14	7	7	11	8	9
T-NOT	23	5	0	17	11	0	17	11	0	15	7	6
T-NOA	28	0	0	21	7	0	23	5	0	19	9	0
T-NMC	17	11	0	14	7	7	14	7	7	11	8	9
T-WMC	21	7	0	15	9	4	14	9	5	13	8	7
T-AMC	22	6	0	16	12	0	17	11	0	14	11	3
Total	128	39	1	98	52	18	99	50	19	83	51	34

Table 8: (Best) class metrics with at least one $|\rho|$ greater than 0.4

Class metric	T-LOC				T-NOT				T-NOA				T_NMC				T-WMC				T-AMC			
	N	L	B	M	N	L	B	M	N	L	B	M	N	L	B	M	N	L	B	M	N	L	B	M
LOC	**0.53**	**0.66**	**0.66**	**0.72**	**0.35**	**0.50**	**0.49**	**0.58**	**0.30**	**0.36**	**0.36**	0.39	0.47	0.59	0.59	0.67	**0.37**	**0.53**	**0.53**	0.60	0.33	0.47	0.45	0.53
LOCCOM	0.49	0.60	0.61	0.65	0.29	0.42	0.43	0.49	0.26	0.31	0.32	0.35	0.41	0.51	0.53	0.59	0.31	0.45	0.46	0.52	0.31	0.42	0.42	0.48
NBI	0.45	0.59	0.61	0.67	0.33	0.48	0.47	**0.58**	**0.30**	**0.36**	0.35	**0.40**	0.46	0.58	0.59	0.67	0.35	0.51	0.52	0.60	**0.34**	**0.48**	**0.47**	**0.56**
NMC	0.44	0.57	0.59	0.63	0.32	0.45	0.45	0.54	0.24	0.30	0.29	0.33	0.46	0.57	0.58	0.65	0.34	0.49	0.50	0.57	0.31	0.43	0.42	0.50
NMCI	0.43	0.52	0.53	0.55	0.29	0.39	0.39	0.45	0.23	0.28	0.28	0.30	0.44	0.52	0.53	0.57	0.32	0.43	0.43	0.48	0.33	0.42	0.41	0.45
NMCE	0.34	0.46	0.48	0.53	0.26	0.39	0.38	0.47	0.19	0.24	0.22	0.27	0.36	0.46	0.47	0.54	0.28	0.42	0.43	0.50	0.22	0.33	0.32	0.40
NPM	0.25	0.41	0.40	0.47	0.26	0.40	0.38	0.47	0.26	0.32	0.31	0.34	0.28	0.40	0.39	0.47	0.23	0.40	0.39	0.46	0.13	0.29	0.27	0.35
RFC	0.45	0.61	0.62	0.68	0.33	0.49	0.47	**0.58**	0.24	0.31	0.29	0.34	**0.48**	**0.61**	**0.61**	**0.69**	0.35	**0.53**	**0.53**	**0.62**	0.31	0.46	0.43	0.53
WMC	0.39	0.55	0.54	0.61	**0.35**	0.49	0.48	0.57	0.28	0.34	0.33	0.37	0.42	0.54	0.53	0.62	0.35	0.51	0.51	0.59	0.22	0.37	0.35	0.44
CBO	0.36	0.42	0.44	0.45	0.17	0.25	0.27	0.31	0.04	0.08	0.09	0.10	0.35	0.41	0.43	0.45	0.23	0.31	0.33	0.36	0.25	0.31	0.32	0.34
CE	0.35	0.41	0.43	0.44	0.14	0.23	0.24	0.28	0.01	0.05	0.06	0.07	0.33	0.40	0.42	0.44	0.21	0.29	0.31	0.34	0.27	0.33	0.33	0.35
CAM	−0.32	−0.48	−0.47	−0.54	−0.28	−0.43	−0.41	−0.51	−0.21	−0.27	−0.26	−0.30	−0.36	−0.48	−0.47	−0.55	−0.29	−0.45	−0.45	−0.53	−0.18	−0.33	−0.30	−0.40

NOF, NPM, RFC and WMC) and two test-effort metrics (T-LOC and T-NOT). Our and their results agree that NMC (which they call FOUT), LOC (which they call LOCC), RFC and WMC highly correlate with both T-LOC and T-NOT. Their results also report that NPM is highly correlated with T-NOT for 4 out of 5 subjects, and that LCOM and NOF are highly correlated with T-LOC for one subject. Our results disagree with such a finding.

Recently, Toure et al. [59] performed a similar study considering seven C&K metrics (CBO, LCOM, WMC, RFC, DIT, NOC, LOC) and three of the test effort metrics considered in our study: T-LOC, T-NOA and T-NMC (which they called TINVOKE). Our and their results agree that LOC, RFC and WMC and CBO highly correlate with test effort. However, they report that also LCOM manifests high correlation. Our results disagree with such a finding.

In a nutshell, our experimental results indicate that the OO design properties size, complexity and coupling most affect the unit testability of Java classes, confirming the results and conclusions of previous studies [28]. Such correlations can be explained as follows:

Testability decreases with the increasing of class size, as the higher the number of lines of code and methods, the more test cases a developer needs to write. *Testability decreases with the increasing complexity*, as effective testing has to cover as many execution paths as possible, and the number of paths increases with the increase of complexity. *Testability decreases with the increasing of coupling*, because classes that have a high coupling are using other parts of the system, increasing the possible execution paths [51].

In addition, this paper indicates that also the OO design property Cohesion highly correlates with test effort: *testability decreases with the decreasing of cohesion.* This could be due to cohesion refers to the degree to which the elements of a class belong together [53]. Smaller behaviors are easier to test than larger behaviors.

3.5 Threats to Validity

A possible threat to external validity is that our results do not generalize to other subjects and OO programming languages. We mitigated this threat by considering thousands of Java projects. The size of our study is several order of magnitude larger than similar studies [19, 20, 59]. Repeat our experiments considering a different OO programming language is an important future work.

A possible threat to internal validity is that there might be errors in our tool or scripts that led to wrong results or metric values. We mitigated this threat by (i) building our static analyzer on top of a fully-fledge tool [2]; (ii) selecting a small samples of classes to manually validate the correctness of the metric values; (iii) testing the critical parts of our scripts and tool. Moreover, we released our data and scripts and we welcome external validation [1].

4 RELATED WORK

The various definitions of Software Testability [62] can be classified into two groups [28]: (i) "ease of testing", measured with test-effort metrics (e.g., test size) [19, 20, 55, 59], and (ii) "ease of revealing faults", measured with test-quality metrics (e.g., mutation score and coverage) [6, 24, 37, 40, 64].

In this paper, we comply with the first interpretation, which is the most popular one in literature [28]. However, while we rely on test-effort metrics to measure the ease of testing, we also consider test-quality metrics. Indeed, the key contribution of this paper is to normalize test-effort metrics with test-quality metrics. At the best of our knowledge, this is new to software testability studies.

We now discuss the work on measuring and predicting test effort (which is most closely related to this paper) and test quality. We conclude the section discussing orthogonal testability work.

Measuring and Predicting the Test Effort. Our work is inspired by the studies of Bruntink and Van Deursen on the correlation between the Chidamber and Kemerer (C&K) and test-effort metrics [19, 20]. These studies provide preliminary evidence that C&K and test-effort metrics correlates, ad thus the C&K metrics can be use to predict the test effort [19, 20].

Badri and Toure studied the correlation of cohesion metrics (LCOM and LCOM*) with test effort [7]. By analyzing two software projects they concluded that it exists a moderate correlation [7]. However, our results do not confirm such a finding. Subsequently, Badri and Toure improved their previous study, increasing the number of software metrics (by adding LOC, CBO, DIT, NOC, WMC and RFC), and considering three projects instead of two [8]. Badri et al. also investigated the effect of control flow to the test effort [9].

Recently, Toure et al. investigated the use of a metric called "Quality Assurance Indicator" to predict test effort on eight open-source Java projects [59]. Gupta et al. proposed a fuzzy technique to combine values of OO software metrics into a single value called testability index [30]. Singh et al. relied on neural networks to predict testing effort from OO metrics [55].

These studies suffer from three main limitations. First, they involve at most eight software systems. Thus, it is difficult to guarantee that the results generalize to other systems. Conversely, in our study we analyzed classes from 1,186 projects. Notably, a small number of subject systems is common to all testability studies [28]. Among the 182 testability studies that involve the analysis of software systems, Garousi et al. showed 161 (88 %) analyze five or less systems, while the remaining 21 (12 %) studies involve at most 45 systems [28]. Second, they focus on some subsets of the class metrics that our study considered. To the best of our knowledge our study is the most comprehensive in terms of number of class metrics. Third, all of these studies do not consider test-quality metrics for normalization. Lacking a quality assessment of the tests leads to imprecise correlation results and prediction models.

Measuring and Predicting the Test Quality. Cruz and Eler analyzed four open-source systems to study the correlation between the Chidamber and Kemerer (C&K) metrics and the quality of the tests (coverage and mutation score) [24]. They concluded that the C&K metrics CBO, LCOM, RFC and WMC have a moderate influence on test quality, and thus a design with low coupling, low complexity, and high cohesion can lead to high coverage and mutation scores.

Khoshgoftaar et al. proposed the use of neural networks to predict testability based on mutation analysis from source code metrics [40]. Jalbert et al. predicted mutation scores by using source code metrics combined with coverage information [37]. Yu et al. proposed a new set of metrics for concurrent programs to predict the mutation score of concurrent tests [64]. Zhang et al. proposed a machine learning approaches to predict mutation score from easy-to-access features, e.g., coverage information, oracle information, and code complexity [65]. Recently, Mao et al. extended the approach of Zhang et al. by considering a cross-project setting, more features and more powerful deep learning models [41].

These studies have a different goal from our work as they aim to predict test quality. Conversely, our work studies the measurement and prediction of test effort using test-quality metrics (mutation score and coverage) as normalization factors.

Orthogonal Work. Besides measuring and predicting test effort, researchers have tackled orthogonal testability goals: design for testability, improvement of testability, and fault proneness.

Design for testability aims at measuring software testability early in the development process, for example during the requirement analysis phase [13, 39, 61], and design [12, 14–16, 50, 60] stages. Applying testability analyses early in the development process has the advantage that design refactoring can improve testability before the implementation starts [50].

Improvement of testability aims at refactoring a program to increase its testability. For example, by obtaining a version of the program that is more amenable to test generation [32, 35, 46, 47].

Basili et al. found that several C&K metrics are associated with fault proneness [11]. Similarly, Gyimothy et al. employed machine learning methods to predict faults from C&K metrics [31]. Briand et al. explored the relationship between class metrics and the probability of fault detection [18]. Yu et al. examined the relationship between class metrics and the fault proneness [63].

All of these approaches have a different goal with the one of in this paper. An interesting future work is to investigate if our idea of normalizing test effort with test quality can also help these approaches to achieve other testability goals.

5 CONCLUSIONS

This paper proposed a new approach to software testability. The new approach extends current approaches by introducing the novel idea of normalizing test effort with respect to test quality. It also presented the results of an extensive study that involves 9,861 pairs of Java classes (with a total of 1,594,309 lines of code) and corresponding JUnit test cases taken from 1,186 GitHub projects.

Our results indicate that normalizing test effort with test quality largely increases the correlation between class metric and test effort. An improved correlation between class metric and test effort means a better prediction of test effort. Indeed, the normalization procedure that we presented in this paper enables the construction of large-scale prediction models from heterogeneous software systems implemented with different test adequacy criteria. Leveraging our normalization procedure we could train different machine learning models considering different versions of our data-set obtained by normalizing test effort by different *target test-quality values*, e.g., 70 %, 80 %, 90 % line coverage. Given in input a class, its class metrics values, and a *target test-quality* value, we could predict the test effort using the prediction model corresponding to the *target test-quality* value in input. We leave the investigation of such an approach as an important future work.

In this paper, we introduced the normalization process under the assumption of a proportional growth of test effort with respect to test quality. For example, if five test cases (T-NOT = 5) achieves a branch coverage of 50 %, our normalization assumes that we need ten test cases (T-NOT = 10) to achieve a branch coverage of 100 %. Another important future work is to study the impact of this assumption on the correlation between class and test-effort metrics.

ACKNOWLEDGMENTS

This work is partially supported by the Swiss SNF project *ASTERIx: Automatic System TEsting of inteRactive software applIcations* (SNF 200021_178742).

REFERENCES

[1] "Expertimental data of this paper," http://bit.ly/30uQoCk
[2] "CKJM-extended," https://github.com/mjureczko/CKJM-extended, last accessed: January 2020.
[3] "JaCoCo Java Code Coverage Library - EclEmma," https://www.jacoco.org, last Accessed: January 2020.
[4] "Metrics - CKJM-extended," http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html, last accessed: September 2019.
[5] "PIT Mutation Testing," http://pitest.org, last accessed: January 2020.
[6] N. Alshahwan, M. Harman, A. Marchetto, and P. Tonella, "Improving Web Application Testing using Testability Measures," in *Proceedings of the International Symposium on Web Systems Evolution*, WSE '09, 2009, pp. 49–58.
[7] L. Badri, M. Badri, and F. Toure, "An Empirical Analysis of Lack of Cohesion Metrics for Predicting Testability of Classes," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 2, pp. 69–85, 2011.
[8] M. Badri and F. Toure, "Empirical Analysis of Object-Oriented Design Metrics for Predicting Unit Testing Effort of Classes," *Journal of Software Engineering and Applications*, vol. 5, no. 7, p. 513, 2012.
[9] M. Badri and F. Toure, "Evaluating the Effect of Control Flow on the Unit Testing Effort of Classes: An Empirical Analysis," *Advances in Software Engineering*, 2012.
[10] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.
[11] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.
[12] B. Baudry and Y. L. Traon, "Measuring Design Testability of a UML Class Diagram," *Information & Software Technology*, vol. 47, no. 13, pp. 859–879, 2005.
[13] B. Baudry, Y. L. Traon, and G. Sunyé, "Testability Analysis of a UML Class Diagram," in *8th IEEE International Software Metrics Symposium* (METRICS 02), 2002, p. 54.
[14] B. Baudry, Y. L. Traon, G. Sunyé, and J. Jézéquel, "Towards a 'Safe' Use of Design Patterns to Improve OO Software Testability," in *12th International Symposium on Software Reliability Engineering* (ISSRE 01), 2001, pp. 324–331.
[15] B. Baudry, Y. L. Traon, G. Sunyé, and J. Jézéquel,"Measuring and improving design patterns testability," in *9th IEEE International Software Metrics Symposium* (METRICS 03), 2003, p. 50
[16] R. V. Binder, "Design for Testability in Object-Oriented Systems," *Communications of the ACM*, vol. 37, no. 9, pp. 87–101, 1994.
[17] R. V. Binder, "Object-Oriented Software Testing," *Communications of the ACM*, vol. 37, no. 9, pp. 28–30, 1994.
[18] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the Relationships Between Design Measures and Software Quality in Object-Oriented Systems," *JSS*, vol. 51, no. 3, pp. 245–273, 2000.
[19] M. Bruntink and A. Van Deursen, "Predicting Class Testability Using Object-oriented Metrics," in *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, (SCAM '04), 2004, pp. 136–145.
[20] M. Bruntink and A. van Deursen, "An Empirical Study Into Class Testability," *JSS*, vol. 79, no. 9, pp. 1219–1232, 2006.
[21] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transaction on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
[22] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 2013.
[23] G. W. Corder and D. I. Foreman, *Nonparametric Statistics for Non-Statisticians*. John Wiley & Sons, Inc., 2011.
[24] R. C. da Cruz and M. M. Eler, "An Empirical Analysis of the Correlation Between CK Metrics, Test Coverage and Mutation Score," *in Proceedings of the 19th International Conference on Enterprise Information Systems* (CEIS '17), 2017, pp. 341–350.
[25] R. B. D'agostino, A. Belanger, and R. B. D'Agostino Jr, "A Suggestion for Using Powerful and Informative Tests of Normality," *The American Statistician*, vol. 44, no. 4, pp. 316–321, 1990.
[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
[27] M. Fewster and D. Graham, *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Publishing Co., 1999.
[28] V. Garousi, M. Felderer, and F. N. Kilicaslan, "A Survey on Software Testability," *Information & Software Technology*, vol. 108, pp. 35–64, 2019.
[29] P. K. Goyal and G. Joshi, "QMOOD Metric Sets to Assess Quality of Java Program," in *the International Conference on Issues and Challenges in Intelligent Computing Techniques*, (ICICT 14), 2014, pp. 520–533.
[30] V. Gupta, K. Aggarwal, and Y. Singh, "A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability," *Journal of Computer Science*, vol. 1, no. 2, pp. 276–282, 2005.
[31] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.
[32] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability Transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, January 2004.
[33] J. Hauke and T. Kossowski, "Comparison of Values of Pearson's and Spearman's Correlation Coefficients on the Same Sets of Data," *Quaestiones geographicae*, vol. 30, no. 2, pp. 87–93, 2011.
[34] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
[35] R. M. Hierons, M. Harman, and C. Fox, "Branch-coverage Testability Transformation for Unstructured Programs," *Comput. J.*, vol. 48, no. 4, pp. 421–436, 2005.
[36] J. R. Horgan, S. London, and M. R. Lyu, "Achieving Software Quality with Testing Coverage Measures," *Computer*, vol. 27, no. 9, pp. 60–69, 1994.
[37] K. Jalbert and J. S. Bradbury, "Predicting Mutation Score Using Source Code and Test Suite Metrics," in *Proceedings of the International Workshop on Realizing AI Synergies in Software Engineering*, (RAISE '12), 2012, pp. 42–46.
[38] M. Jureczko and D. Spinellis, *Using Object-Oriented Design Metrics to Predict Software Defects*, Monographs of System Dependability. 2010, Models and Methodology of System Dependability, pp. 69–81.
[39] M. Khan and R. Srivastava, "Flexibility: A Key Factor To Testability," *International Journal of Software Engineering & Applications*, vol. 6, no. 1, p. 89, 2015.
[40] T. M. Khoshgoftaar, E. B. Allen, and Z. Xu, "Predicting Testability of Program Modules Using a Neural Network," in *Proc. of the IEEE Symp. on Application-Specific Systems and Soft. Engineering Technology*, (ASSET '00), 2000, pp. 57–62.
[41] D. Mao, L. Chen, and L. Zhang, "An Extensive Study on Cross-Project Predictive Mutation Testing," in *Proceedings of the IEEE International Conference on Software Testing, Validation and Verification*, (ICST '19), 2019, pp. 160–171.
[42] R. Martin, "OO Design Quality Metrics : an Analysis of Dependencies," *ROAD 1995*, vol. 2, no. 3, 1995.
[43] M. Mattsson, "Effort Distribution in a Six Year Industrial Application Framework Project," in *Proceedings of the IEEE International Conference on Software Maintenance, 1999.* (ICSM '99), 1999, pp. 326–333.
[44] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.
[45] P. McMinn, "Co-testability Transformation," in *Evolutionary Test Generation*, 2008.
[46] P. McMinn,"Search-based Failure Discovery Using Testability Transformations to Generate Pseudo-Oracles," in *Proceedings of the Genetic and Evolutionary Computation Conference*, (GECCO 19), 2009, pp. 1689–1696.
[47] P. McMinn, D. W. Binkley, and M. Harman, "Empirical Evaluation of a Nesting Testability Transformation for Evolutionary Testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 3, pp. 11:1–11:27, 2009.
[48] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007.
[49] J. C. Miller and C. J. Maloney, "Systematic Mistake Analysis of Digital Computer Programs," *Communications of the ACM*, vol. 6, no. 2, pp. 58–63, 1963.
[50] S. Mouchawrab, L. C. Briand, and Y. Labiche, "A Measurement Framework for Object-Oriented Software Testability," *Information & Software Technology*, vol. 47, no. 15, pp. 979–997, 2005.
[51] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. McGraw-Hill, Inc., 2010.
[52] N. M. Razali, Y. B. Wah *et al.*, "Power Comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests," *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.
[53] R. W. Sebesta, *Concepts of Programming Languages*. Boston: Pearson,, 2012.
[54] Y. Singh and A. Saha, "Predicting Testability of Eclipse: A Case Study," *Journal of Software Engineering*, vol. 4, no. 2, pp. 122–136, 2010.
[55] Y. Singh, A. Kaur, and R. Malhotra, "Predicting Testing Effort Using Artificial Neural Network," in *World Congress on Engineering and Computer Science*, (WCECS '08), 2008, pp. 1012–1017.
[56] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An Empirical Study on Object-Oriented Metrics," in *Proceedings of the 6th International Symposium on Software Metrics*, (METRICS '99), 1999, pp. 242
[57] F. Toure, M. Badri, and L. Lamontagne, "A Metrics Suite for Junit Test Code: a Multiple Case Study on Open Source Software," *Journal of Software Engineering Research and Development*, vol. 2, no. 1, p. 14, 2014.
[58] F. Touré, M. Badri, and L. Lamontagne, "Towards a Unified Metrics Suite for Junit Test Sases," in *The 26th International Conference on Software Engineering and Knowledge Engineering* (SEKE 13), 2014, pp. 115–120.
[59] F. Toure, M. Badri, and L. Lamontagne, "Predicting Different Levels of the Unit Testing Effort of Classes Using Source Code Metrics: A Multiple Case Study on

Open-source Software," *Innovations in Systems and Software Engineering*, vol. 14, no. 1, pp. 15–46, 2018.

[60] Y. L. Traon, F. Ouabdesselam, and C. Robach, "Analyzing Testability on Data Flow Designs," in *11th International Symposium on Software Reliability Engineering* (ISSRE 00), 2000, pp. 162–173.

[61] Y. L. Traon and C. Robach, "Testability Measurements for Data Flow Designs," in *4th IEEE Inter. Software Metrics Symposium*, (METRICS 97), 1997, pp. 91–98.

[62] J. M. Voas and K. W. Miller, "Software Testability: The New Verification," *IEEE software*, vol. 12, no. 3, pp. 17–28, 1995.

[63] P. Yu, T. Systa, and H. Muller, "Predicting Fault-proneness Using OO Metrics. An Industrial Case Study," in *ECSMR*. IEEE, 2002, pp. 99–107.

[64] T. Yu, W. Wen, X. Han, and J. H. Hayes, "Predicting Testability of Concurrent Programs," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, (ICST '16), 2016, pp. 168–179.

[65] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2019.