# A Large-Scale Study on the Usage of Testing Patterns that Address Maintainability Attributes

## Patterns for Ease of Modification, Diagnoses, and Comprehension

Danielle Gonzalez *, Joanna C.S. Santos*, Andrew Popovich*, Mehdi Mirakhorli*, Mei Nagappan†
*Software Engineering Department, Rochester Institute of Technology, USA
†David R. Cheriton School of Computer Science, University of Waterloo, Canada
{dng2551,jds5109,ajp7560, mxmvse}@rit.edu, mei.nagappan@uwaterloo.ca

***Abstract*—Test case maintainability is an important concern, especially in open source and distributed development environments where projects typically have high contributor turnover with varying backgrounds and experience, and where code ownership changes often. Similar to design patterns, patterns for unit testing promote maintainability quality attributes such as *ease of diagnoses*, *modifiability*, and *comprehension*. In this paper, we report the results of a large-scale study on the usage of four xUnit testing patterns which can be used to satisfy these maintainability attributes. This is a first-of-its-kind study which developed automated techniques to investigate these issues across 82,447 open source projects, and the findings provide more insight into testing practices in open source projects. Our results indicate that only 17% of projects had test cases, and from the 251 testing frameworks we studied, 93 of them were being used. We found 24% of projects with test files implemented patterns that could help with maintainability, while the remaining did not use these patterns. Multiple qualitative analyses indicate that usage of patterns was an ad-hoc decision by individual developers, rather than motivated by the characteristics of the project, and that developers sometimes used alternative techniques to address maintainability concerns.**

***Index Terms*—Unit Testing, Maintenance, Open Source, Mining Software Repositories, Unit Test Patterns, Unit Test Frameworks**

## I. Introduction

The *unit test* serves as an important facet of software testing because it allows individual "units" of code to be tested in isolation. With the development of automated testing frameworks, such as the *xUnit* collection, writing and executing unit tests has become more convenient. However, writing quality unit tests is a non-trivial [7], [23] task. Similar to production code, unit tests often need to be read and understood by different people. The moment a developer writes a unit test, it becomes legacy code that needs to be maintained and evolve with changes in production code.

Particularly, writing quality unit tests that encourage *maintainability* and *understandability* is a very important consideration for open source projects and the distributed development environment in general, where projects have high contributor-turn over, contributors have varying backgrounds and experience, and code ownership changes often [25].

Existing research and best practices for unit testing focus on the quality of the test's design in terms of optimizing the coverage metric or defects detected. However, recent studies [5], [10], [12], [21] emphasize that the maintainability and readability attributes of the unit tests directly affect the number of defects detected in the production code. Furthermore, achieving high code coverage requires *evolving* and *maintaining* a growing number of unit tests. Like source code, poorly organized or hard-to-read test code makes test maintenance and modification difficult, impacting defect identification effectiveness. In "xUnit Test Patterns: Refactoring Test Code" George Meszaro [18] presents a set of automated unit testing patterns, and promotes the idea of applying patterns to test code as a means of mitigating this risk, similar to how design patterns are applied to source code.

Previous studies [5], [14], [15], [20] have looked at test case quality in open source and industry, studying the effects of test smells and bad practices, and detecting quality problems in test suites. However, these and other testing-related works [17], [33] are limited to a small number of projects and have not taken into account use of testing frameworks and patterns that can be used to address maintainability attributes.

In this paper, we aim to assess the satisfaction of test maintainability quality attributes in open source projects by studying the use of automated unit testing frameworks and the adoption of four of Meszaros' testing patterns within a large number of projects.

First, we conducted a large-scale empirical study to measure the application of software testing in the open source community, providing insights into whether open source projects contain unit tests and which testing frameworks are more common. Second, to evaluate the maintainability characteristics of unit tests written in the open source community, we have defined three quality attributes: *Ease of Modification*, *Ease of Diagnoses*, and *Ease of Comprehension*. To assess the maintainability characteristics of the tests with regard to these quality attributes, we detect the adoption of four unit testing patterns that, when applied to unit tests within open source projects, help satisfy these attributes. Our novel approach includes a data set of 82,447 open source projects written in 48 languages, and our approach for identifying pattern use is language-agnostic, increasing the generalization of our results to open source development as a whole. We also conduct a qualitative study to reveal project factors which may have influenced a developer's decision whether or not to apply

patterns to their tests.

More precisely, the following motivating question and sub-questions have been explored and answered in this study: **Motivating Question:** *Do Test Cases in Open Source Projects Implement Unit Testing Best Practices to Satisfy Test Maintainability Quality Attributes?* To accurately answer this question, we investigate the following sub-questions:

- **RQ1:** What percentage of projects with tests applied patterns that address *Ease of Modification*?
- **RQ2:** What percentage of projects with tests applied patterns that address *Ease of Diagnoses*?
- **RQ3:** Do automatically generated test plans and checklists focused on testing breaking points improve the security and reliability of a software product?

Despite investigating the use of only four testing patterns, our novel approach to measuring test quality will highlight important facts about the quality of test cases present in open source projects.

The reminder of this paper is organized as follows. A description of related works is provided in section II, and section III defines important terms used throughout the paper. Section IV describes in detail the methodology for answering our three research questions. Results are presented in section V. Qualitative analyses and discussion of our pattern results is presented in section VI. Threats to validity are acknowledged in section VII, and we end with our conclusions and proposed future work.

## II. Related Work

In this section we identify relevant work studying unit testing in open source software, maintainability and quality of unit test cases, and the relationship between production and test code.

The studies most closely related to the work presented in this paper were presented by Kochar et. al. This group was the first to conduct a large-scale study on the state of testing in open source software [15] using 20,817 projects mined from Github. They studied the relationship between the presence and volume of test cases and relevant software project demographics such as the number of contributing developers, the project size in lines of code (LOC), the number of bugs and bug reporters. They also measured the relationship between programming languages and number of test cases in a project. A year later, they conducted a quality-focused study [16] using code coverage as a measure of test adequacy in 300 open source projects.

Other small-scale studies have evaluated the quality or effectiveness of unit tests in software projects. Badri et. al's used two open source Java projects to study levels of testing effort to measure the effectiveness of their novel quality assurance metric [3]. Bavota et. al [5], [6] performed two empirical analyses to study the presence of poorly designed tests within industrial and open source Java projects using the JUnit framework, and the effects these bad practices have on comprehension during maintenance phases. Evaluations of the effectiveness of automated unit test generation tools show the importance of the usability and readability quality attribute, such as a study by Rojas et. al [24] on automatic unit tests generated by*Evosuite*. To improve understandability of test cases Panichella et. al created a summarization technique called *Test Describer*. [22]. Athanasiou et. al [2] created a model to measure test code quality based on attributes such as maintainability found to positively correlate with issue handling.

Some works have taken an approach opposite to the one presented in this paper to measure the quality of unit tests by measuring the presence of 'test smells' and bad practices within projects. These works uses the 'test smells' defined by Van Deursen et. al [28] and broadened by Meszaros [18]. Van Rompaey and Demeyer [30] defined a metric-based approach to detect two test smells, and later Brugelmans and Rompaey [8] developed *TestQ* to identify test smells in test code. To reduce erosion of test maintainability over time due to fixture refactoring, Greiler et. al [14], developed *TestHound*, which detects test smells. Neukirchen et. al [20] have used the presence of test smells to identify maintenance and reusability problems in test suites which use TCCN-3.

A number of works have been published which study the relationships between production code and unit tests. Using graphs obtained from static and dynamic analysis, Tahir and MacDonell [27] applied centrality measures to quantify the distribution of unit tests across five open source projects in order to identify the parts of those projects where the most testing effort was focused. Zaidman et. al [33] use coverage and size metrics along with version control metadata to explore how tests are created and evolve alongside production code at the commit level on two open source projects. Van Rompaey and Demeyer [29] have established conventions for improving the traceability of test code such as naming conventions, static call graphs, and lexical analysis. These methods have been used in other papers to [27] establish heuristics for identifying tests in source code.

In this study we expand the scope of existing explorations of testing in open source by increasing the number of projects and languages being examined. The novel contributions presented in this paper are the use of three quality attributes, measured via pattern and test framework adoption in projects, as a measure of unit test maintainability in the open source community.

## III. Background and Definitions

We begin by defining the important concepts and terms required to understand our work. In this section we define the metrics, three maintenance quality attributes for unit tests, and the testing patterns used throughout the paper.

### A. Project Metrics

*Testing Ratio:* To better understand the quantity of test code present in open source projects, we used the *TPRatio* metric, defined as the Production Code to Unit Test Code Ratio:

$$TPRatio = \frac{UnitTestLOC}{ProductionLOC}$$

TPRatio measures the ratio of unit test lines of code to production (non-test) lines of code. Other works have used code coverage to measure test volume in projects [32], but we chose the TPRatio metric as a reasonable alternative because we do not have a reliable means of collecting coverage data for all the languages in our dataset. Ward Cunningham provides a discussion of this metric on his website [9] which mentions it as a 'quick check' before measuring coverage. This metric has also been used in a previous study by Kochar et. al [15].

*Project Size* Project size within our dataset varies widely, which can impact how certain results, such as TPRatio, are presented. To prevent this, we divide projects in our dataset into four sizes when presenting results:

- Very Small: Project Size $<$ 1 kLOC
- Small: 1 kLOC $\leq$ Project Size $<$ 10 kLOC
- Medium: 10 kLOC $\leq$ Project Size $<$ 100 kLOC
- Large: Project Size $\geq$ 100 kLOC

### B. Unit Test Quality Attributes

To evaluate the maintainability characteristics of unit tests written in the open source community, we have defined the following quality attributes, derived from literature [5], [18], [28] that can impact the quality of unit tests. These attributes, summarized below, were used as driving requirements to identify patterns that improve the quality of unit tests.

- QA#1 - Ease of Modification: The ability of unit tests in a system to undergo changes with a degree of ease.
- QA#2 - Ease of Diagnoses: The measure of difficulty when attempting to track and inspect a test failure.
- QA#3 - Ease of Comprehension: The degree of understandability of a unit test being read by a developer not originally involved in its writing.

### C. Testing Patterns

'Test smells', introduced by Van Deursen et. al [28], negatively affect the maintainability of unit tests [5], [6]. Guidelines for refactoring test code to remove these smells have been proposed [11], [28] but in order to *prevent* smells and promote high quality tests, Meszaros has detailed 87 goals, principles, and patterns for designing, writing and organizing tests using the *xUnit* family of unit testing frameworks [18], [19] Therefore, in order to evaluate the maintainability of unit tests in open source projects, we chose to measure the adoption of some of these patterns. We developed 3 criteria when choosing patterns; first, a pattern must address at least one of the quality attributes defined in section III-B. Next, a pattern must be optional for a developer to implement, i.e. not enforced by the framework. This will help us to understand the decisions open source developers make. Finally, it must be possible to detect the use of the pattern or principle using a programming-language-agnostic heuristic, because we needed to be able to detect their use in test files written in numerous languages. After careful review, we chose four patterns that met the selection criteria. Table I provides a brief mapping between each pattern and the quality attributes they address as well as rationale for satisfying the quality attributes based on existing works. Patterns are described in brief Coplin form [1] in the context of test smells/quality goals below. Our detection heuristics are described in section IV-C.

Simple Tests: **Context:** A test with multiple assertions. **Problem:** Obscure Test(Eager Tests) [18], [28] If a test verifies multiple conditions and one fails, any conditions after it will not be executed. After a failure, a test with multiple assertions is more difficult to diagnose, and any checks appearing after the failing assertion would not be executed. **Solution:** A *simple test* is a test method which only checks one code path of the subject under test. Meszaros categorizes simple tests as a goal of test automation, which can be realized by following the *Verify One Condition Per Test* principle.

Implicit Teardown: **Context:** Objects and fixtures are created within a test class or method to simulate the right test environment. **Problem:** Conditional Test Logic (Complex Teardown) [18]. It is difficult to ensure that test fixtures have been removed properly if the teardown code is difficult to comprehend, and leftover fixtures can affect other tests. While most objects can be automatically destroyed in languages with garbage-collection, some objects need to be destroyed manually such as connections or database records. **Solution:** In this pattern, a `teardown` method containing removal code for items that are not automatically removed is run after each test method execution, no matter if it passes or fails. In a class with many tests, this pattern eases maintenance by ensuring *all* persisting items are removed, and promotes readability by making it easier for developers to understand which test items are being destroyed, and when.

Assertion Message: **Context:** Tests written with xUnit based frameworks verify conditions using *Assertion Methods*. By default, these methods do not return any descriptive information when they fail. **Problem:** Assertion Roulette [28] In a test with multiple assertions, or when many tests are run simultaneously, this lack of output can make it difficult to diagnose which assertion failed. **Solution:** This problem can be mitigated by adding an *Assertion Message* to each assertion method. While this might seem like a trivial pattern, its presence in an assertion shows a conscious choice to improve the readability of the test. This pattern can also be used as an alternative solution to the problem discussed in *Simple Test*.

Testcase Class Per Class (TCCPC) **Context:** a project with a large number of classes. **Problem:** Poor Test Organization. *Where do I put a new test?* There are a number of options, with the worst-case being ALL project test cases in ONE test class. **Solution:** Developers who take this into consideration early in a project may wish to create one *Testcase Class per (Source) Class* being tested. This improves readability and maintenance while making organization more straightforward. This pattern can also be applied to a mature project with existing tests, but if it contains a very large test suite, this refactoring can be time consuming.

### D. Testing Frameworks

Instead of manually writing and executing complex custom tests, developers can use a unit testing framework, which

TABLE I: Overview of Testing Patterns Studied

| | Simple Test | I.Teardown | Assertion M. | Rationale for Satisfying the Quality Attributes |
|---|---|---|---|---|
| **QA1. Ease of Modification** | | [16] | | Implicit teardown improves internal test structure by keeping track of all resources created in a test and automatically destroy/free when a teardown method is called. TCCPC organizes tests in relation to production code, placing all the test methods for one source class into one Testcase Class. |
| **QA2. Ease of Diagnossis** | [16] [24] | | [4] [5] | Defect traceability and test readability are improved when only one condition is checked per test. Assertion messages help identify which assertion method failed while providing useful context. |
| **QA3. Ease of Comprehension** | [4] [5] | [16] | [4] [5] | Organizing the destruction of test resources, verifying one condition per test, and augmenting assertions with appropriate messages enhance the understandability of each test. |

provides syntax conventions and utility methods. Presence of a test framework is one criteria we use to identify test files, described in section IV-B. Mezaros' [18] collection of test patterns are based on the *xUnit* family of unit test frameworks (JUnit, CUnit, etc.). However, since there are usually multiple test frameworks for each programming language, we look for both xUnit and other frameworks.

## IV. Methodology

To answer our motivating question, our methodology consists of three steps: (i) retrieve a large number of open source projects from online repositories, (ii) identify test files across these projects, and (iii) detect testing patterns across test files. We discuss each of these steps in the following subsections.

The data used in this study include 82,447 open source projects, 251 unit testing frameworks, and 4 unit testing patterns. A full copy of this data can be found in our project website [1]. To detect test files and pattern use in the projects, we developed automated, heuristic-based techniques.

### A. Gathering Open Source Projects

Our first step was to extract a large-scale collection of software projects from open source repositories. Our current collection contains 82,447 projects extracted from GitHub. To retrieve projects, we used GHTorrent [13], which provides Github's public data and events in the form of MongoDB data dumps containing project metadata such as users, commit messages, programming languages, pull requests, and follower-following relations. We also used *Sourcerer* [4], which provided version-ed source code from multiple releases, documentation (if available), and a coarse-grained structural analysis. After extracting projects from these resources, we performed a data cleanse in which we removed all the empty projects and duplicate forks. Table II shows the number of projects collected per programming language.

### B. Extracting and Analyzing Test Cases

Next, we developed a heuristical approach to identify test files among other source files and identify which automated testing frameworks were used to implement the tests. Our automated test case extraction and analysis tool contains the following components:

[1]All the collected data and the tools developed in this paper can be found at: https://goo.gl/Mc7tHk

TABLE II: Number of Projects included in the study

| Language | Freq. | Language | Freq. | Language | Freq. |
|---|---|---|---|---|---|
| JavaScript | 20201 | Lua | 343 | F# | 21 |
| Java | 14493 | Haskell | 338 | Processing | 21 |
| Ruby | 8558 | Rust | 241 | Dart | 19 |
| Python | 7180 | CoffeeScript | 230 | FORTRAN | 17 |
| PHP | 6889 | Matlab | 214 | Objective-C++ | 16 |
| C++ | 4758 | Groovy | 161 | Haxe | 15 |
| C | 4369 | Puppet | 99 | Julia | 15 |
| C# | 4365 | TypeScript | 63 | Elixir | 13 |
| Shell | 1879 | Emacs Lisp | 62 | Pascal | 13 |
| Objective-C | 1493 | PowerShell | 55 | Prolog | 13 |
| Go | 858 | Arduino | 46 | Visual Basic | 13 |
| Swift | 782 | ASP | 44 | Common Lisp | 12 |
| Scala | 633 | OCaml | 29 | Mathematica | 11 |
| VimL | 440 | Assembly | 27 | AGS Script | 9 |
| Perl | 399 | Erlang | 27 | BlitzBasic | 9 |
| Clojure | 386 | ActionScript | 26 | Scheme | 9 |
| R | 366 | D | 23 | Cuda | 8 |

*Total number of projects: 82,447

Catalog of Automated Testing Frameworks This catalog contains a list of existing testing frameworks and regular expressions which represent the import statements for each framework. To establish this catalog, we started with a precompiled list of unit testing frameworks sorted by language from Wikipedia [31]. For each framework in this list we then manually collected additional information such as applicable languages. Most importantly, we searched source code, official documentation, and tutorials to find the framework's API and code annotations in order to determine how each framework is imported into a project. With this information we developed heuristics for determining which (if any) framework a project was using. Frameworks which required payment for use, without source code or sufficient documentation, or for which we could not develop a technique to identify its import statement were then excluded from the list. Our final list contained 251 testing frameworks for 48 different languages.

Test Case Detection and Analysis This component used two *heuristical conditions* to analyze each project in order to determine they contained test files. Our analysis used a *voting mechanism* to determine if each condition had been met. The conditions were as follows:

**Condition #1** `The file imported at least one automated unit testing framework`

For each test framework we used the list of specific import statement(s) to create regular expressions, and pattern matching was used to identify the import statement within the source code. The accuracy of this approach was evaluated using a peer review process including 50 randomly selected projects.

With 100% accuracy, this approach identifies various forms of imported testing libraries.

**Condition#2** `The file contained keywords commonly used by frameworks in the files programming language`

We also curated a list of common *testing keywords* and *constructs* (such as `assert` and `verify`) found in each programming language (Table III.). This list was established by manually analyzing at least 10 sample test cases per language, and collecting *framework-specific terms* from the documentation of each testing framework.

We processed every file in our collection and calculated a binary value for both condition. A file received a vote of 0 for a condition if it did not meet that particular condition, or a 1 if the file did meet the condition. A file was considered a test file if both conditions received a vote of 1.

This heuristical approach to test case identification differs from most existing methods in the literature [15], [29] such as selecting files which include the word 'test' in the file name. We chose not to include this condition to avoid relying on file name conventions because we believe the use of frameworks and test-keywords are stronger indications of test files and will not exclude files that do not follow the 'testX' naming convention. We also excluded other conditions such as Fixture Element Types or Static Call Graphs [29] in order to develop language-agnostic heuristics which could be applied to all the languages in our dataset.

TABLE III: Sample of Testing Keywords Per Language

| Language | Tag Regex |
|---|---|
| Java | @Test, @Before, @After, @AfterClass, @BeforeClass, @Ignore, @RunWidth, @Parameters, assert.* |
| C#softw | [TestClass], [TestMethod], [ClassInitialize], [TestInitialize], [TestCleanUp], [TestFixture], [SetUp], [Test], [ExpectedException(typeof(.*))], [Fact], [Theory], [InlineData(.*)], [CollectionDefinition(.*)], [Collection(.*)] |
| JavaScript | test(, describe(rb,assert(, assert_, describe |
| Python | def test_, testmod( |
| C | START_TEST, END_TEST, setUp(, tearDown(, UnityBegin(, RUN_TEST(, TEST_ASSERT_EQUAL(, cmocka_unit_test(, assert.*, VERIFY( |
| C++ | START_TEST, END_TEST, BOOST_CHECK(, BOOST_REQUIRE(, CPPUNIT_TEST(, CPPUNIT_ASSERT_EQUAL(, CPPUNIT_FAIL( |
| Perl | ok(, is(, isnt(, like(, unlike(, cmp_ok(, pass(, fail( |
| PHP | TEST, FILE, EXPECT, EXPECTF, SKIPIF, INI, CLEAN, test, assert, @test |

### C. Detecting Patterns in Test Files

To detect the selected patterns within test files we developed another set of heuristical techniques. For each pattern, we were able to identify unique characteristics such as code structure, test method signature, and reserved keywords which indicate the presence of said pattern.

The *Implicit Teardown* and *Assertion Message* patterns are built-in but *optional* components of the xUnit framework, and can be best identified in source code by verifying the presence of certain keywords. The heuristical approach used to detect each pattern is described in the following:

- *Assertion Messages* are passed as arguments into an `assert` method in a test. Common forms for an assertion with a message include: `assertCONDITION(ARG1, ARG2, [MESSAGE])`, `assert(ASSERT, CONDITION, [MESSAGE])`, and `assertBOOLEAN(ASSERT, [MESSAGE])`. By developing regular expressions for these three method signatures we can detect if an assert function contains a message. We totaled the number of matches for each signature variation in every file, and considered a test case if there was at least one match.
- *Simple Test* is detected by tracking the number of asserts in a test case. If there is only one assert, we consider it a simple test. We adopted the previous heuristic to detect this pattern.
- *Implicit Teardown* required a semantic heuristic *and* a relatively more complex search to verify its presence. Our heuristic first finds a `teardown` method, a built-in signature in testing frameworks used to implement the teardown pattern. After finding this method signature in a test method, our heuristic searches for 'remove' or 'destroy' methods to indicate that items are being deleted. In our heuristic, the method name 'teardown' *must* be found, along with at least least one of the two other methods.
- *Testcase Class Per Class* is identified by first creating a call graph between all classes in the project, including test classes. This call graph is used to check if there is one test class which depends on a source class in production code. We used an existing commercial tool [26] to generate the call graph. This tool can only perform static analysis on a limited number of languages. Therefore, we were only able to detect this pattern in 11 of the 48 languages present in our dataset: C, C++, C#, FORTRAN, Java, Pascal, VHDL, HTML, CSS, PHP, and Javascript.

### D. Validation of Data Extraction Techniques

To validate the correctness of the data collected, we performed the following checks:

*Test Files:* To verify the correctness of the test files, we randomly selected 20 projects and manually identified any test files and any test framework the projects imported. We then ran our test case analysis (conditions 1 & 2) on these files and verified that the results matched. Our approach achieved an accuracy of 100%.

*Pattern Files:* To verify the correctness of the pattern files, we randomly selected and manually verified files which were identified by our heuristic to contain a pattern, 15 per pattern. Our pattern detection approach had 100% accuracy.

## V. Results

This section presents the results for the main motivating question and the three consequent research questions.

### A. The State of Software Testing in Open Source

The following data provides general insights into the state of software testing in open source, specifically project size, test-to-code ratios, and framework usage across the projects

TABLE IV: Test-Production Code Ratios Across Project Sizes

| | All Proj. | Very Small | Small | Medium | Large |
|---|---|---|---|---|---|
| **Total # Projects** | 82447 | 38379 | 25425 | 13322 | 5321 |
| **Avg. TPRatio** | 0.06 | 0.05 | 0.06 | 0.08 | 0.08 |
| **Max. TPRatio** | 12.20 | 12.20 | 11.05 | 7.92 | 4.62 |
| **#Proj. With Tests** | 15119 | 3207 | 4203 | 4131 | 3578 |

studied. This provides better context and background for the interpretation of our research question results.

Distribution of Project Sizes: As mentioned in section III-A, we grouped the 82,447 projects in our collection into four size categories (*very small*, *small*, *medium* and *large*) to better understand how testing varies with project size. After grouping the projects, we found that our collection was 46.55% *very small* projects, 30.82% *small* projects, 16.16% *medium* projects and 6.45% *large* projects.

Test Code to Production Code Ratio: We observed that 14,153 (17.17%) of projects included test files. We computed the ratio of lines of code from test cases to total number of line in the production code (TPRatio III-A) and report this metric for each project size category in Table IV.

The average TPRatio of all the projects in our collection was 0.06. We also note that *small* and *very small* projects included fewer test files. Out of 38,379 *very small* projects and 25,425 *small* projects, only 3,207 (8.4%) and 4,203 (16.5%) of projects had at least one test file, respectively. *Medium* and *large* projects in our dataset contained more tests (31% and 67.2%, respectively).

**Testing In Open Source:** 17.17% of projects studied contained test files. The average ratio of test code to production code was very low, 0.06. Smaller projects had a lower ratio than larger projects. One hypothesis for this result is that medium and large projects are more complex and, therefore, introduce automated testing practices more to ensure code correctness.

Test Framework Adoption in the Open Source Community: Of the 251 unit testing frameworks we searched for in 48 programming languages, we detected usage of 93 frameworks in 9 languages (C, C++, C#, Java, JavaScript, Perl, PHP, Python and Ruby). Table V presents the three most-used frameworks per language. In the case of the Perl language, we only detected the usage of two testing frameworks.

Relying only on the total number of projects that adopted a framework to identify the most commonly used ones would result in a bias. As presented in Table II, the distribution of the programming languages over the projects in our repository is not uniform. Therefore, the frameworks that are used in the most frequent languages in our repository would more likely be placed at the top of the ranking. To overcome this, we normalized the frequency of the frameworks by dividing the *framework frequency* by the *total number of projects in the framework's programming language(s)*. Table VI enumerates the ten most used frameworks sorted by this normalization value. From this table we observe that `Mocha`, a testing framework for JavaScript projects, is the most used framework.

TABLE V: Top 3 Unit Testing Frameworks per Language

| | Framework | # Occurrences |
|---|---|---|
| C | Check | 251 |
| | GLib Testing | 98 |
| | SimpleCTest | 38 |
| C++ | Boost Test Library | 113 |
| | Google Test | 29 |
| | Google C++ Mocking Framework | 29 |
| C# | NUnit | 443 |
| | Moq | 164 |
| | Rhino Mocks | 30 |
| Java | JUnit | 3841 |
| | Mockito | 573 |
| | TestNG | 303 |
| JavaScript | Mocha | 6714 |
| | JSUnit | 758 |
| | Enhance JS | 737 |
| Perl | Test::More | 49 |
| | Test::Builder | 2 |
| PHP | PHPUnit | 1000 |
| | PHP Unit Testing Framework | 867 |
| | SimpleTest | 348 |
| Python | unittest | 1663 |
| | Autotest | 466 |
| | Nose | 336 |
| Ruby | Test::Unit | 202 |
| | Shoulda | 75 |
| | minitest | 56 |

TABLE VI: Top 20 Most Used Frameworks

| Framework | Language | Freq. | Normalized Freq. |
|---|---|---|---|
| Mocha | JavaScript | 6714 | 0.33 |
| JUnit | Java | 3841 | 0.27 |
| unittest | Python | 1663 | 0.23 |
| PHPUnit | PHP | 1000 | 0.14 |
| PHP Unit Testing | PHP | 867 | 0.13 |
| Test::More | Perl | 49 | 0.12 |
| NUnit | C# | 443 | 0.10 |
| Autotest | Python | 466 | 0.06 |
| Check | C | 251 | 0.06 |
| SimpleTest | PHP | 348 | 0.05 |
| Nose | Python | 336 | 0.05 |
| Mockito | Java | 573 | 0.04 |
| Moq | C# | 164 | 0.04 |
| JSUnit | JavaScript | 758 | 0.04 |
| Enhance JS | JavaScript | 737 | 0.04 |
| Sinon.js | JavaScript | 638 | 0.03 |
| py.test | Python | 213 | 0.03 |
| Chai | JavaScript | 523 | 0.03 |
| Vows | JavaScript | 517 | 0.03 |
| Boost Test Library | C++ | 113 | 0.02 |

Normalized Freq= Framework Freq. Divided by the Total #Projects in that Programming Language

**Test Framework Usage:** From 251 testing frameworks included in our study, only 93 were imported by developers in open source projects. Mocha (Javascript) was the most commonly used framework.

### *B. Satisfaction of Quality Attributes through Pattern Adoption*

We identified 3,363 projects that contain at least one pattern (23.76%) and 11 projects which contained all four patterns (0.08%). Table VII shows the number of test files and projects which implement each pattern. *Assertion Message* was the most commonly adopted pattern, occurring in 49,552 test files among the 399,940 test files found in projects in our collection. The other patterns were not as widely adopted. *Testcase Class Per Class* was adopted in 4,565 test files, *Simple Test* was only applied in 1,614 test files, and *Implicit Teardown* was used in 920 test files. Therefore, few test cases applied patterns which can impact maintainability attributes.

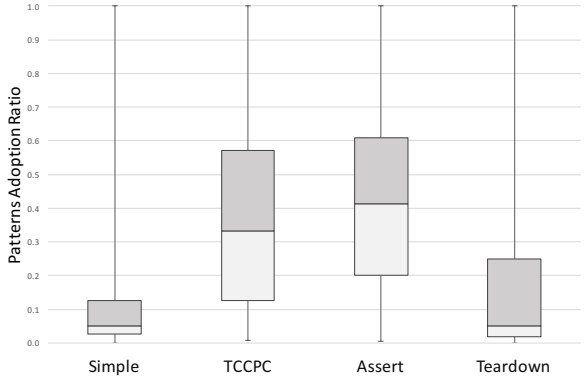
All further analysis focuses on projects with tests containing patterns.

TABLE VII: Pattern Detection Results

| Pattern | # Files | # Projects |
|---|---|---|
| Assertion Message | 49552 | 3048 |
| Simple Test | 1614 | 531 |
| Implicit Teardown | 920 | 153 |
| TCCPC | 4565 | 810 |

Pattern Adoption Ratio: In order to better understand *how often* the patterns were adopted in projects, we computed the adoption ratio of the patterns as the *number of test case files with the pattern* divided by the *total number of test files* in the project. This ratio was computed per pattern for the projects that adopted each pattern. The adoption ratios of patterns are presented in Figure 1. From this figure, we find that both *Assertion Message* and *TCCPC* had the highest adoption rates, which means that these patterns were consistently applied throughout the projects' test cases. In fact, the median adoption rates for *Assertion Message* and *TCCPC* were 41.2% and 33.3%, respectively. Conversely, the adoption ratios for *Simple Test* and *Implicit Teardown* were both 5%.

Fig. 1: Pattern Adoption Ratios (Excludes No-Pattern Projects)




Pattern Adoption And Number of Test Cases: In addition to the analysis of the raw results, we wanted to examine if patterns are typically found in projects containing a lot of test cases or not. Therefore, we built a linear regression model to examine the relationship between the adoption ratio of each pattern (independent variables) to the number of test cases (dependent variable) in the corresponding project. However, since the number of test cases could just be a function of the size of a project, we include the size of the project (in KLOC) as a control variable in the regression model.

From Table VIII, we find that as expected there is a strong positive relationship between size of the project and the number of test cases. However, surprisingly, we notice that the relationship between adoption ratio and the number of test cases is negative. This implies that smaller projects are more disciplined in using test case patterns.

**Pattern Adoption:** Open source projects do not frequently adopt the studied testing patterns. Smaller projects have fewer test files and test-to-production code ratios, but a higher frequency of pattern adoption compared to larger projects.

TABLE VIII: Linear Regression Coefficients Projects

| | Estimate | Std. Error | t value | $Pr(>\|t\|)$ |
|---|---|---|---|---|
| (Intercept) | 7.285e+01 | 4.780e+00 | 15.243 | $<$ 2e-16 |
| Simple Test | -1.373e+02 | 3.191e+01 | -4.303 | 1.73e-05 |
| TCCPC | -7.658e+01 | 1.104e+01 | -6.938 | 4.71e-12 |
| Assertion Message | -5.872e+01 | 8.391e+00 | -6.998 | 3.09e-12 |
| Teardown | -9.289e+01 | 4.200e+01 | -2.212 | 0.027 |
| TotalLinesOfCode | 2.079e-05 | 1.754e-06 | 11.849 | $<$ 2e-16 |

To answer our three research questions, we used a QA-Pattern mapping (See Table I) to relate the patterns we chose to study with the quality attributes they best represent. From the 3,363 projects which included patterns, for each quality attribute we calculated the percentage of projects that implemented each mapped pattern as well as all mapped patterns. This data is shown in Table IX. Note that percentages are based *only* on the total number of projects with patterns.

***RQ1:*** *What percentage of projects with patterns addressed the Ease of Modification Quality Attribute?*

The testing patterns which can satisfy the *Ease of Modification* quality attribute for tests are *Implicit Teardown* and *Testcase Class Per Class*. 143 (4.32%) projects applied *Implicit Teardown*, improving modifiability of internal test contents. 815 (23.02%) projects implemented *Testcase Class Per Class (TCCPC)*, improving the modifiability of test structure in relation to production code. 41 (1.16%) of projects implemented both patterns, addressing both internal and structural test modifiability.

**Ease of Modification:** Approximately one-fourth of the projects with patterns implemented the *Testcase Class per Class* pattern, providing a maintainable design for test classes and their associated production classes. Fewer projects adopted *Implicit Teardown*, which helps structure the destruction of test fixtures.

***RQ2:*** *What percentage of projects with patterns addressed the Ease of Diagnosis Quality Attribute?*

To satisfy *Ease of Diagnoses*, we looked for adoption of *Simple Test* and *Assertion Message* patterns. Both patterns improve defect traceability and test readability. From all projects that implement patterns, 576 projects (16.27%) implemented *Simple Test*, and 3,048 projects (86.02%) implemented *Assertion Message*. 473 (13.36%) projects implemented both, strongly improving defect traceability and test readability. Although these two patterns are intuitive to enforce, a large number of projects have not implemented them.

**Ease of Diagnoses:** Despite benefits in diagnoses of the *Simple Test* and *Assertion Message* patterns, less than one fourth of projects adopted both or *Simple Test* without *Assertion Message*. However, over three fourths of projects used *Assertion Message* without *Simple Test*.

TABLE IX: Percentage of Projects Satisfying Each Maintainability Quality Attributes Using Testing Patterns

| Ease of Modification | | | Ease of Diagnosis | | | Ease of Comprehension | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TCCPC | Teardown | TCCPC & Teardown | Simple Test | Assertion Message | Simple Test & Assertion Message | Simple Test | Assertion Message | Teardown | Assertion Message & Simple Test | Simple Test & Teardown | Assertion Message & Teardown | All |
| 23.02 | 4.32 | 1.16 | 16.27 | 86.08 | 13.36 | 16.27 | 86.08 | 4.32 | 11.92 | 0.03 | 4.80 | 1.44 |

***RQ3:*** *What percentage of projects with patterns addressed Ease of Comprehension Quality Attribute?*

Comprehension improvement involves a combination of test modifiability and diagnosis improvements. The testing patterns which can satisfy *Ease of Comprehension* in our study were *Implicit Teardown*, *Assertion Message*, and *Simple Test*. As individual usage of these patterns has been reported above, we will now discuss combinations of these patterns. 422 (11.92%) projects implemented *Assertion Message* and *Simple Test*, 170 (4.80%) projects implemented *Assertion Message* and *Implicit Teardown*, and 1 (0.03%) project implemented *Simple Test* and *Implicit Teardown*. 51 (1.44%) implemented all three patterns.

**Ease of Comprehension:** Comprehension can be addressed with three of the patterns studied, but less than 2% of projects with patterns implemented all three. The most implemented combination was *Assertion Message* and *Simple Test* (11.92%), which are more intuitive to implement than *Implicit Teardown*.

From the results reported for our three RQs, we can see that the quality attribute addressed in the highest number of projects is *Ease of Diagnoses*. This indicates that developers most often applied the patterns which could help them identify why a test has failed. Relatively, *Ease of Modification* and *Ease of Comprehension* were not satisfied across many open source projects. To further investigate these findings, we report three qualitative analyses on projects with different pattern usage in the following section.

## VI. Qualitative Analyses

In addition to the quantitative results reported in the previous section, we have conducted three qualitative studies to understand differences between projects with different levels of pattern adoption. First we investigated whether there are any additional project characteristics which influenced pattern adoption. Next, we studied the contributions of developers to the test cases with pattern within projects which applied all four patterns. Finally, we examined test files from projects with no patterns to determine if they used other approaches to address test maintainability.

### *Analysis #1: Pattern Adoption and Project Characteristics*

To identify potential correlations between test pattern adoption and other project characteristics, we collected project artifacts such as source code, online repositories, and documentation for 29 randomly selected projects from 3 categories: 9 projects that applied all 4 patterns, 10 projects that applied 1, 2, or 3 (*Some*) patterns, and 10 that applied 0 (*No*) patterns. We looked at test documentation, organization, 'coverage', industry sponsorship, and number of contributors per file.

Project Selection: All projects in our dataset were sorted by randomly generated numbers. 10 projects from each category (*All*, *Some*, and *No* patterns) were then randomly chosen from the sorted list. To ensure that projects were suitable for our study, we developed exclusion criteria. Selected projects were verified that: (1) there was at least one online artifact available such as a Github page, (2) contributor information was available, and (3) the project contained original source code. For projects with *Some* patterns, we also tried to choose projects with different combinations of patterns. If a project failed to meet these criteria, it was excluded from the study and a new project was randomly selected using the process above. Of the 11 total projects in our dataset which applied all 4 patterns, 2 had to be excluded, so we were only able to study 9.

Analysis Results:

**Project Demographics:** For all 29 projects selected, the number of contributors ranges from 2 to 620, and the number of forks ranges from 0 to 3,108. 16 unit testing frameworks were used, and projects were written in Java, Javascript, C#, C, C++, Python, Scala, and PHP.

**Test Documentation & Rules:** Our first question was whether the presence of test-specific documentation affected pattern adoption. Therefore, we manually searched available artifacts to see if projects had any guidelines (documentation) or templates specifically for unit testing code. We also wanted to know if projects had rules for new contributions, specifically that existing tests must pass, and new code requires new unit tests.

Projects with *All* patterns required the developers to add new tests alongside source code contributions more frequently than the projects with *Some* or *No* patterns. Projects with *Some* patterns most often required that existing tests run and pass (6) and also included more guidelines for writing tests (4). We found instances of unit test template code in 2 projects, one with *All* patterns and the other with *No* patterns.

**Test Organization & 'Coverage':** We examined if projects with well-organized tests were more likely to adopt patterns. To measure how well tests were organized in a project, one author searched its directories to find where all test files were located. Project test organization was ranked *Great*, *Good*, *OK*, or *Poor* based on how easily all tests in the project could be located and if they were organized by component. Due to the range of languages used, we were unable to quantitatively measure coverage, so we considered 'coverage' in terms of how many project components had *any* tests. For test organization, only 1 project (*All*-pattern) received a *Great* rating, but projects with *Some* and *No* patterns had equal amounts of *Great*(0), *Good*(7), and *Poor*(1) ratings. Surprisingly, projects with *All* patterns had the highest amount

of *Poor*ly organized projects (2). For 'coverage', more projects with *Some* patterns had tests for all components (7), and projects with *No* tested more than half of components (4).

**Industry Contribution:** Company sponsorship of a project is another characteristic which could potentially affect pattern adoption. We also noted if the 'core contributors' of a project (identified through acknowledgments and Github metrics) worked for the sponsoring company. Of the 12 projects with industry sponsors, 4 applied *All* patterns, 2 applied *Some* patterns, and 6 applied *No* patterns. 8 projects had rules for unit testing (2 *All*, 2 *Some*, and 4 *No*) and 6 included test specific guidelines or templates (1 *All*, 1 *Some*, and 4 *No*). 2 projects had *Poor* test organization (1 *All*, 1 *No*), 9 had *Great* or *Good* organization (4 *All*, 1 *Some*, 4 *No*), and 1 (*No* patterns) had *OK* organization. The majority of industry projects had high 'coverage', with only 1 (*Some*) project testing less than half of its components.

**# Contributors Per Test File:** Finally, we were interested in a possible correlation between pattern adoption and the number of contributors who work on an individual test file. Across all projects, the number of contributors to a single test file ranged from 1 to 10. Projects with *All* or *Some* pattern usage did not have higher contributors per test file than projects with no pattern usage. The average for all projects was 1 to 2 contributors per file.

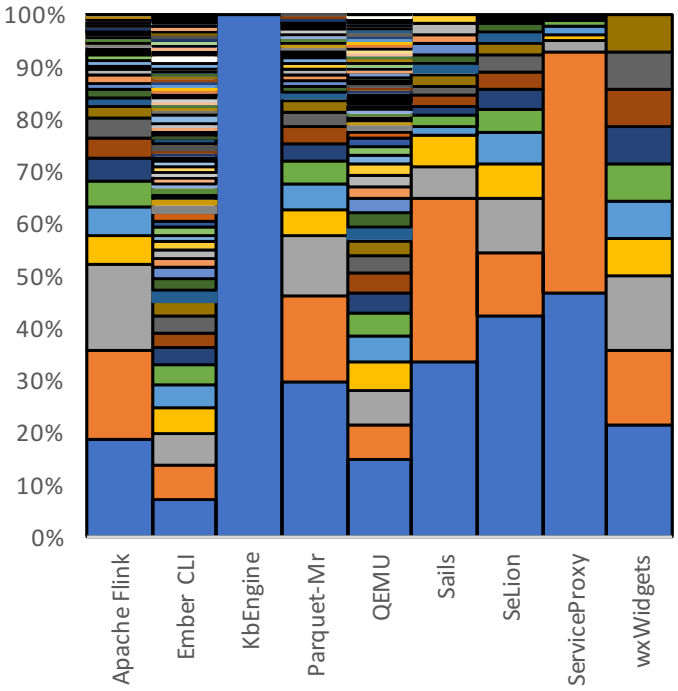
Analysis Conclusions: The only characteristic found most frequently in projects with *All* patterns was requiring new tests to be added to contributions, and one instance of *Great* test organization. Projects with *Some* and *No* patterns were more similar than projects with *All* and *Some* patterns in their inclusion of these characteristics as well. Interestingly, the differences in occurrences of these characteristics across all project categories was very small. It does appear, however, that projects with industry sponsors often addressed test organization, testing rules & guidelines, and coverage. However, only 6 of these projects implemented *All* or *Some* patterns. Because we were unable to identify a strong influence on pattern adoption in this study, we performed another qualitative analysis on the contributors to projects using *All* patterns.

*Analysis #2: Contributors to Projects Using All Patterns*

Next, we investigated the influence of developers on the adoption of testing patterns. For this analysis, we identified the developers that contributed to test files in the 9 projects with *All* patterns. To do this, we retrieved the commit logs of these projects and identified the developers that committed code to each test file with a pattern and computed the *total* number of testing files with patterns that *each* developer contributed to.

Analysis Results: By observing the total number of test pattern files each developer contributed to, we found that a *fewer* number of developers were contributing to *most* of the testing files with patterns. Figure 2 shows the proportion of contributions to testing files with patterns made by each developer in the projects. In this figure, each color represents one developer that contributed to a testing pattern within the project. From this, we observe that in almost all projects between 2 to 3 developers are contributing the most to test files with patterns. For KbEngine, there was only one developer that was contributing to testing patterns, who is also the creator and maintainer of the project. For Ember Cli, only 9 of 100 total developers who wrote test cases were contributing most of the test files.

Fig. 2: Proportion of Contributions to Test Files with Patterns Across the Investigated Projects that Implemented All Patterns (Each Color is a Contributor)




Analysis Conclusions: We observed that fewer developers were contributing to the majority of test files with patterns. This suggests that testing patterns adoption in a project is an *ad-hoc* personal decision made by individual developers rather than a project-wide effort to improve test quality.

*Analysis #3: Tests in Projects Using No Patterns*

Our final analysis was an investigation into test cases from projects which did not use patterns. Since these files contain no patterns, our goal was to see what, if any, other techniques were used to satisfy *ease of modifiability, diagnosis, and comprehension*. 5 test files from each of the 10 projects without patterns used in the first qualitative study were randomly selected (50 total files) containing a total of 219 test methods. Our search was focused on solutions which implemented test failure feedback (*Assertion Message*) for ease of diagnoses and fixture destruction (*Implicit Teardown*) for ease of modifiability, but we also looked for characteristics which would improve comprehension of the tests such as naming conventions and the presence of comments.

Analysis Results: The first general observation from this study was that within a project, files were usually consistent in their use of naming conventions and comment usage, which helped with *comprehension* of the test cases. This was not the

case in only 3 projects. Only 34% of files used comments to describe the behavior or purpose of a unit test. 54% of projects used clear, natural language naming conventions for file, method and variable names. 38% of files (written in Javascript) used an alternative naming convention called `describe`, a special method where the functionality of the test is passed as a parameter. The remaining 8% of files used sequential numbering.

As an alternative to *Implicit Teardown*, 74% of test files relied solely on garbage collection and the remaining 26% of projects used custom methods (such as `cleanup`). 82% of the files used an alternative to *Assertion Message*: Most Javascript files studied used an alternative assertion library called 'Should', which uses an `it('should...')` method syntax rather than an assertion (38%). On failure, this message is displayed. 26% of files used exception throwing for errors, 6% used try/catch blocks, 4% used custom logging objects, and the remaining 26% used no failure feedback at all.

Analysis Conclusions: From these observations it is clear that the projects with no patterns use other techniques besides the XUnit patterns to address ease of maintainability and diagnoses, such as readable naming conventions and alternative failure feedback techniques. However, the data also shows that these projects seldom use comments to describe test behavior, which would improve comprehension. A future study to investigate these alternative techniques may reveal interesting results.

**Qualitative Conclusions:** Pattern adoption is independent of project characteristics, and it is dependent on individual developers. Project without patterns instead used existing libraries and proprietary and primitive techniques to satisfy quality attributes.

## VII. Threats to Validity

The following threats to the validity are identified.

### A. Construct Validity

We assumed that all test cases written for a project were included in the project's source code repository. Thus, if zero test files were detected, we say the project does not contain tests. Our approach detected the *presence* of test files in the project but not how or if they were run, so we do not know if the tests are still relevant. However, this is of low concern because we are more interested in the *quality* of the tests present in the project.

### B. Internal Validity

When mining software repositories, it is important to consider that some repositories may not contain projects, or may be a 'fork' of another project. To prevent repeated data in our results, we removed all empty and duplicate project forks. For duplicates, we removed all but the original project *or* the oldest fork if we did not have the original project in our dataset.

### C. External Validity

The 82,447 projects studied do not provide a complete perspective of the entire open source community. Projects may have custom tests, or use a testing framework which which was not included in this study. However, the large volume of projects and frameworks included allow new insights into the current state and quality of open source testing. Also, while the use of only four testing patterns was investigated, we believe their use is a good indication that developers took quality into consideration when designing their tests. We do **not** claim that a project without the patterns studied does not address quality as it is possible that other patterns or techniques were applied.

### D. Reliability Validity

We used heuristical methods (defined in section IV) to detect test files and patterns. This approach is similar to those used in other studies (see section II), but there are related risks. It is possible that some test frameworks were not included in our detection process because there are multiple ways to import a framework into a project. To reduce this risk, we maually searched all available documentation for each framework to find import statements. We also used a list of reserved testing keywords in our detection tool. Further, while we searched for all possible ways of implementing the patterns within the frameworks considered, it is possible that we missed instances of pattern use if the implementation did not match our heuristics. Finally, the static analysis tool [26] used to generate the file dependency data for detecting the Testcase Class Per Class pattern does not recognize all of the languages in our dataset. Therefore, our results for this pattern only included the projects which we were compatible with Understand. In the future we would like to detect use of this pattern in all the languages in our dataset.

## VIII. Conclusion

In this paper we have performed a large-scale empirical analysis on unit tests in 82,447 open source software projects written in 48 languages. We found 14,153 (17%) projects containing test files written in 9 languages using 93 test frameworks. 3,363 (24%) projects applied at least one of the 4 patterns, and 11 projects applied all 4 patterns. *Ease of Diagnoses* was the most commonly addressed quality attribute, and *Assertion Message* was the most adopted pattern. We also found that although smaller projects contain fewer test files than larger projects, they apply patterns more frequently. Through three qualitative analyses of the projects with and without patterns, we found that pattern adoption is an ad-hoc decision by individual project contributors. In summary we find that open source projects often do not adopt testing patterns that can help with maintainability attributes. More research is needed to understand why and how we can help developers write better test cases that can be maintained and evolved easily.

## Acknowledgments

This work was partially funded by the US National Science Foundation under grant numbers CCF-1543176.

## REFERENCES


[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. Fault-tolerant telecommunication system patterns. *Rising [45]*, pages 81–94, 1996.

[2] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, 2014.

[3] M. Badri, F. Toure, and L. Lamontagne. Predicting unit testing effort levels of classes: An exploratory study based on multinomial logistic regression modeling. *Procedia Computer Science*, 62:529 – 538, 2015. Proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15).

[4] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[5] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

[6] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. Are test smells really harmful? An empirical study. *Empirical Software Engineering*, 20(4):1052–1094, Aug. 2015.

[7] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 179–190, 2015.

[8] M. Breugelmans and B. Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[9] W. Cunningham, D. Branson, J. Grig, M. Ischenko, D. Knuth, and J. Baum. Production code vs unit tests ratio, Aug 2012.

[10] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 107–118, New York, NY, USA, 2015. ACM.

[11] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[12] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.

[13] G. Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[14] M. Greiler, A. van Deursen, and M.-A. Storey. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331. IEEE, 2013.

[15] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 103–112. IEEE, 2013.

[16] P. S. Kochhar, F. Thung, D. Lo, and J. Lawall. An empirical study on the adequacy of testing in open source projects. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 215–222. IEEE, 2014.

[17] W. Lam, S. Srisakaokul, B. Bassett, P. Mahdian, T. Xie, N. Tillmann, and J. de Halleux. Parameterized unit testing in the open source wild. 2015.

[18] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, Upper Saddle River, NJ, 2007.

[19] G. Meszaros, S. M. Smith, and J. Andrea. The test automation manifesto. In *Conference on Extreme Programming and Agile Methods*, pages 73–81. Springer, 2003.

[20] H. Neukirchen and M. Bisanz. *Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites*, pages 228–243. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[21] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. Automatic test case generation: What if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 130–141, New York, NY, USA, 2016. ACM.

[22] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558, New York, NY, USA, 2016. ACM.

[23] M. Perscheid, D. Cassou, and R. Hirschfeld. Test quality feedback improving effectivity and efficiency of unit testing. In *2012 10th International Conference on Creating, Connecting and Collaborating through Computing*, pages 60–67. IEEE, 2012.

[24] J. M. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 338–349. ACM, 2015.

[25] I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou. Open source software development should strive for even greater code maintainability. *Commun. ACM*, 47(10):83–87, Oct. 2004.

[26] Scitools. Understand static analysis tool. https://scitools.com/. (Accessed on 09/19/2016).

[27] A. Tahir and S. G. MacDonell. Combining dynamic analysis and visualization to explore the distribution of unit test suites. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pages 21–30. IEEE, 2015.

[28] A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[29] B. Van Rompaey and S. Demeyer. Establishing traceability links between unit test cases and units under test. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 209–218. IEEE, 2009.

[30] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.

[31] Wikipedia. List of unit testing frameworks — wikipedia, the free encyclopedia, 2016.

[32] T. W. Williams, M. R. Mercer, J. P. Mucha, and R. Kapur. Code coverage, what does it mean in terms of quality? In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, pages 420–424, 2001.

[33] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.