Computing and Informatics, Vol. 32, 2013, 1001–1030, V 2021-Jan-25

AUTOMATING TEST CASE IDENTIFICATION IN OPEN SOURCE PROJECTS ON GITHUB

Matej Madeja, Jaroslav Porubän, Michaela Bačíková, Matúš Sulír, Ján Juhár, Sergej Chodarev, Filip Gurbáľ

Department of Computers and Informatics
Technical University of Košice
042 00 Košice, Slovakia
e-mail: matej.madeja@tuke.sk, jaroslav.poruban@tuke.sk, michaela.bacikova@tuke.sk, matus.sulir@tuke.sk, jan.juhar@tuke.sk, sergej.chodarev@tuke.sk, filip.gurbal@tuke.sk

Abstract. Software testing is one of the very important Quality Assurance (QA) components. A lot of researchers deal with the testing process in terms of tester motivation and how tests should or should not be written. However, it is not known from the recommendations how the tests are actually written in real projects. In this paper the following was investigated: (i) the denotation of the test word in different natural languages; (ii) whether the test word correlates with the presence of test cases; and (iii) what testing frameworks are mostly used. The analysis was performed on 38 GitHub open source repositories thoroughly selected from the set of 4.3M GitHub projects. We analyzed 20,340 test cases in 803 classes manually and 170k classes using an automated approach. The results show that: (i) there exists weak correlation ($r = 0.655$) between the word *test* and test cases presence in a class; (ii) the proposed algorithm using static file analysis correctly detected 95% of test cases; (iii) 15% of the analyzed classes used `main()` function whose represent regular Java programs that test the production code without using any third-party framework. The identification of such tests is very low due to implementation diversity. The results may be leveraged to more quickly identify and locate test cases in a repository, to understand practices in customized testing solutions and to mine tests to improve program comprehension in the future.

Keywords: Program comprehension, Java testing, testing practices, test smells, open source projects, GitHub

Mathematics Subject Classification 2010: 68-04

1 INTRODUCTION

Software quality is a highly desirable feature of a software product that is delivered by verification and validation activities. One of Quality Assurance (QA) components is software testing as a popular risk management strategy to ensure that source code meets all the requirements. However, the development of such tests is a time-consuming and costly process, as it represents more than a half of the entire development process [28]. At first glance, the task of creating tests is straightforward — to test, but in addition to that tests describe the expected behavior of the production code being tested. Years ago, Demeyer et al. [6] suggested that if the tests are maintained together with the production code, their implementation is the most accurate mirror of the product specification and can be considered as up-to-date documentation. Obviously, tests can contain a number of useful production code metadata that can support program comprehension.

Understanding the code is one of the very first tasks a developer must struggle with before the implementation of a particular feature. When the product specification changes (e.g. the requirements for new features are added), the developer must first understand them, then create his/her own mental model [4] and finally, the created mental model is expressed in a specific artifact — code implementation. The problem is that two developers are likely to create two different mental models for the same issue. If more than one developer is working on a project, it is expected that they will think differently and that the same feature will be implemented in different ways depending on the author. A comprehension gap arises when one developer needs to adapt another programmer's mental model from the code, and this is the most time consuming process.

An assumption can be made that by using the knowledge about the structure and semantics of tests and their connection to the production code, it is possible to increase the effectiveness of program comprehension and reduce the comprehension gap. This would be possible, for example, by enriching the source code with metadata from the tests directly into the production code, e.g. data used for testing, test scenarios, objects relations, comments, etc. To achieve this goal, it is necessary to know in detail how the tests are actually written and what kind of data they encompass.

There exist many guidelines on how tests should be created. First, naming conventions may aid the readability and comprehension of the code. According to the empirical study by Butler et al. [3], developers largely follow naming conventions. It can be assumed that these conventions will also be followed for the test code. Our previous research [23] shows that there is a relation between the naming of identifiers in the test code and the production code being tested. This indicates that the relationship between the test and production code is not only at the level

of method calls, object instances, or identifier references, but also at the vocabulary level that is connected to the domain knowledge and mental models of a tester and a developer.

Furthermore, many authors [25, 21, 9] define best practices to simplify the test with the benefit of a faster understanding of the testing code and the identification of test failure. Some guidelines lead to avoiding test smells [32] because as reported by recent studies [26, 29], their presence might not only negatively affect the comprehension of test suites but can also lead to test cases being less effective in finding bugs in the production code. All mentioned approaches are only recommendations, but do not really express how the tests are written in real projects. That means we know how tests should be written, but we do not know how they are written in practice. Many researchers have tried to clarify the motivation of writing tests [22, 1, 18], the impact of test-driven development (TDD) on code quality [8, 2] or the popularity of testing frameworks [33]. According to our knowledge, there is no research that deals with the analysis of tests in independent projects.

To reveal testing practices in real and independent projects it is necessary to find a way to identify test cases in a project, without the time-consuming code analysis. Much more important than the number of test cases is the information where they are located. When a testing framework is used, the test identification is mostly straightforward, e.g. by the presence of the framework imports. On the other hand, it is advisable to consider tests that do not use any third-party framework and can be regarded as customized testing solutions. Considering the ways in which test cases were identified in related work and based on the authors' experience of Java test cases development, it can be assumed that there is a relation between the word *test* and the number of test cases in a file. That means searching for the *test* string could be beneficial for faster test case identification. Based on the previous reasoning, this paper defines the following hypotheses:

H1: There is a correlation between the occurrence of the word *"test"* in the file content and the number of test cases.

H2: Tests are usually constituted by means of test frameworks, but there are also other ways of automated code testing.

These hypotheses represent only a partial step towards mining information from tests to support program comprehension. Upon successful identification, it will be possible to analyze the impact of testing frameworks on test writing and developer thinking. The knowledge about how tests are written can be used to mine information from suitable testing code segments to enrich the production code with this information and support code comprehension to speed up the development process.

This paper analyzes 38 projects that have been carefully selected from all GitHub projects with a majority of code written in Java language. In addition to confirming or refuting the hypotheses, the paper provides an overview of known test frameworks, examining whether it is appropriate to search for tests using the word "test" due to different natural languages of developers, and provides an algorithm for static code

analysis to automate the identification of test cases. Because testing is a relatively complex task that can be made up of different types of tests (unit, user interface, etc.) and it is not possible to analyze them in a reasonable amount of time, this paper focuses exclusively on unit testing.

Section 2 presents the current state of research and the gaps found in the research. In section 3, the research method is described, including the analysis of the word *test* in different languages, the selection of unit testing frameworks, used search strategies and the process of analysis. This section also presents the pseudocode of the proposed algorithm used for automated test case detection. Section 4 summarizes the results, threats to validity are mentioned in section 5, and conclusions can be found in section 6.

2 STATE OF THE ART

Many researchers examine software testing but man still know little about the structure and semantics of test code. This chapter summarizes the related work of software testing from various perspectives.

2.1 Software testing and quality evaluation

Learning about real testing practices is a constant research challenge. The goal of such research is mostly to find imperfections and risks, learn, and make recommendations on how to prevent them and how to streamline their development. Leitner and Bezemer [20] studied 111 Java-based projects from GitHub that contain performance tests. Authors identify tests by searching for one or more terms in the test file name or for the presence of popular framework import, solely in the `src/test` project directory. Selected projects were subjected to a manual analysis, in which they monitored several metrics. The most important results for this paper was the fact that 103 projects also included unit tests, usually following standardized best practices. On the other hand, the performance testing approach of the same projects often appears less extensive and less standardized. Another finding was that 58 projects (52%) mix performance tests freely with their functional test suite, i.e., performance tests are in the same package, or even the same test file, as functional tests. Six projects implemented tests as usage examples. Using a similar approach we would like to analyze unit tests, but with careful selection from all GitHub projects at a specific time, resulting in more relevant projects used for analysis.

Code coverage, also known as test coverage, is a very popular method for evaluating project quality. Ellims et al. [7] investigated the usage of unit testing in practice in three projects that authors evaluated as well-tested. Statement coverage was found to be indeed a poor measure of test adequacy. According to the findings of Hemmati [12], basic criteria such as statement coverage are a very weak metric, detecting only 10% of the faults. A test case may cover a piece of code but miss its

faults. According to Hilton et al. [13], coverage can be beneficial in the code review process if a smaller part of the project is evaluated. By reducing coverage to a single ratio of the whole project, much valuable information could be lost. Kochhar et al. [16] performed an analysis of 100 large open-source Java projects showing that 31% of the projects have coverage greater than 50% and only 8% are greater than 75%.

Many experiments try to express the quality of tests by testing "mutants" [15], i.e., by modifying a program in small ways to create artificial defects. According to Gopinath et al. [10] mutants do not necessarily represent real bugs, therefore, they are not able to relevantly evaluate the quality of the test suite nor to find relations between the coverage and mutants' reveal. However, there is a statistically significant correlation between code coverage and bug kill effectiveness of real software errors (non-mutants) [17]. The quality of the test suite is influenced by the way the mental model is expressed in the code, so examining real tests is more beneficial instead of using mutants.

The fact that unit tests are the most common test type in a project is confirmed by Cruz et al. [5]: 39% of 1000 analyzed Android projects used unit tests. Another finding was that frequently updated projects were more aware of the importance of using automated tests than those updated several years ago. The adoption of tests has increased over the last years, so focusing on information mining from the tests makes sense.

Another type of research was done by Munaiah et al. [24], who focused on the assessment of GitHub projects. They proposed a tool that can be used to identify repositories containing real engineered software projects. The aim was to eliminate the repository noise such as example projects, homework assignments, etc. One of the metrics they use for assessment is unit test occurrence in the project using test ratio (number of source lines of code in test files to the number of source lines of code in all source files) to quantify the extent of unit testing effort. Package imports of *JUnit* and *TestNG* frameworks were searched to identify tests in the project. This method could be useful when looking for the occurrence of specific testing frameworks in the code.

2.2 Identified issues and improvement opportunities

Based on the mentioned research, it is possible to roughly estimate the motivation of the tester, the relationship of code coverage and test case quality, and best practices improving the reliability of test suites. However, it is still unknown what frameworks are used for Java test projects, whether there are tests that are not dependent on the framework, or whether the particular project uses its own framework. Also the following issues in the existing research have been identified:

1. *Respondent type*: In some cases, research conducted in an academic environment does not provide real testing practices because students are not full-fledged developers; they often have little experience. Tracking students' practices provides skewed data. Observations of practices in a single software company or country

are also misleading because people from the same corporate or state environment have similar practices and they influence each other, e.g., company code style, used frameworks, similar education, etc.

2. *Project type and size*: Experiments often compare test practices on sample projects, which do not represent real projects, but are created or modified for a particular experiment. The size of such projects is very small so they contain only a few lines of production code. Code comprehension of such projects is much simpler and therefore tests are much simpler, too. Real project normally contains more complicated tests.
3. *The amount of analyzed data*: Studies that analyze enterprise or open source projects usually select a small number of projects (mostly up to 5), which results in samples too small for generalized claims.

With respect of the mentioned research imperfections, this paper tries to carefully select projects for analysis from a large number of independent projects to provide as much relevant results as possible.

3 METHOD

First of all, it is necessary to find suitable projects containing test cases. Thus, metadata of all GitHub open-source projects was obtained via GHTorrent [11] (section 3.1) due to their high availability. GHTorrent collects projects' metadata from GitHub, one of the biggest project sharing platform in the world. The experiment was limited to projects whose the most used language is Java. In existing research is common technique searching for testing frameworks' imports [30] or to search for files with the world *test* in the filename [20]. Because our main goal for the future is to improve production code comprehension, it is necessary to find particular test cases and not only a test class.

As we go deeper in this study and try to identify specific test cases (not only test classes), it is necessary to consider whether the searching for the word *test* is appropriate. Keep in mind, that the aim is not to count number of test cases in a project. Otherwise, we could run test via automated build tool (e.g. ant, maven or gradle) and collect the number of tests. In that case, the issue is that building such open source projects often fail [31] and we need to build every single project and run tests what is a time consuming task. In this paper we try to count and especially find the location of such test cases.

Since the testing process can also be denoted as verification, examination, etc., an in-depth analysis (section 3.2) of testing process denotation in various foreign languages was performed, which showed that searching for the word *test* is suitable. Due to the limitations of the GitHub Search API it was possible to search only one word across all Github Java projects.

As the framework is assumed to influence developer thinking and test case implementation, a list of 50 unit testing frameworks for Java (section 3.3) has been

created. Because the goal is to detect customized testing practices compared with framework–based ones in existing projects, it is not possible to use an automated method, and since it is not possible to manually analyze all GitHub projects, we need to select the most suitable ones. Based on the meaning of the word *test* we assume that there will be a correlation between the occurrence of the word *test* (in file content or filename) and the number of test cases. Therefore, three datasets were created using the searching GitHub API for (section 3.4):

1. the word *test* in filename,
2. the word *test* in file content,
3. frameworks' imports in file content (38 frameworks).

Every single project was searched as mentioned above, 4.3 million projects in total. It is possible to expect that the more occurrences of the word *test* in the project, the more test cases will be present in it and the more we will learn from it in the future. Therefore, projects with the highest occurrence of the word test (in file content or filename) or with the highest occurrence of a specific framework's import were selected for manual analysis. Using searching for *test* regardless of the framework, we were also able to analyze testing practices without using any third-party framework, i.e. customized testing solutions. Because GitHub contains many projects that are not relevant, e.g. testing, homework or cloned projects, rules for searching relevant projects have been defined (section 3.4.2), resulting in set of projects used for manual and automated analysis. A script for automated analysis was created to partially automate the identification of test cases and to collect some metrics about particular files (see section 3.5). All methodology details are described in the following sections.

3.1 Data source

To provide conclusions that are as general as possible, it was necessary to choose the most general sample of data possible. It would be ideal to analyze all types of projects, i.e. proprietary and open source. This experiment is focused exclusively on open source projects, for the reasons of access to proprietary projects is limited.

GitHub[1] is a distributed code repository and project hosting web site. It has become one of the most popular web-based services to host both proprietary and mostly open source projects, therefore, we can consider it a suitable source of projects. It provides an open Application Programming Interface (API)[2] allowing one to work with all public projects (with small exceptions).

To avoid the latency of the official API, the GitHub Archive project[3] stores public events from the GitHub timeline and publishes them via Google BigQuery.

[1] https://github.com/
[2] https://docs.github.com/en/rest
[3] https://www.gharchive.org/

Downloading via Google BigQuery is charged. *GHTorrent* [11] was used instead that provides a mirror of GitHub projects' metadata. It monitors the GitHub public event timeline, retrieves contents and dependencies of every event and request GitHub API to store project data into database. That includes general info about projects, commits, comments, users, etc. The study data mining started in May 2019, therefore, the last MySQL dump[4] `mysql-2019-05-01` has been downloaded and imported into our local database.

3.2 Denotation of the word *test*

Leitner et al. [20] searched for tests only in `src/test` directory and test classes identified manually. However, the tests can be placed in any project's directory (e.g. Android[5] uses `src/androidTest`). Another approach is to search for *"test"* string in filenames as executed by Kochhar et al. [18], because they assumed that the tests would be exclusively in files containing the case-insensitive *"test"* string. As in the previous case, best practices lead the developer to use *test* in the file name, but it is not mandatory. For this reason, the most accurate should be searching for the word *test* in the file content. Of course, firstly it is necessary to consider whether the word *test* is the right one for searching.

Therefore, the meaning of the word *test* using Google Translate[6] was verified in 109 different languages (all available by Google) as follows:

1. **From English to foreign language and back to English**
   By means of this method the most frequent[7] meanings of the word *test* in a foreign language were obtained. We obtained multiple meanings per language. By translating them back to English we found out which foreign language translations correspond to the original word *test*.
2. **From foreign language to English and back to foreign language**
   The opposite approach was used to find whether the string *test* has a meaning in particular foreign language. The word was translated into English and all its meanings were verified against the available translation alternatives in the given language.

Multiple translations ensured that the correct meaning of the word in a particular language was understood. Using the 1st method it was found out that word sets related to testing process of different foreign languages are mostly translated as *test* in English, see Figure 1. This means that when a foreign developer would like to express something related to testing (e.g. to write a test case), he/she will use mostly the word *test*. In this meaning it is the first choice when searching test cases

[4] https://ghtorrent.org/downloads.html
[5] https://developer.android.com/
[6] https://translate.google.com/
[7] Frequency determined by Google Translate service, indicates how often a translation appears in public documents: 3 - high; 2 - middle; 1 - low frequency.

by a string. In a very marginal denotation (i.e. the translation was found with low frequency) occurred meaning outside of testing area, e.g., *essay*, *audition* or *flier*. Because such meanings occurred only infrequently, it can be omitted. There were also 14 languages that did not include the word *test* in their reverse translation at all, but its meaning was rather denoting *examination*, *check* or *quiz*.

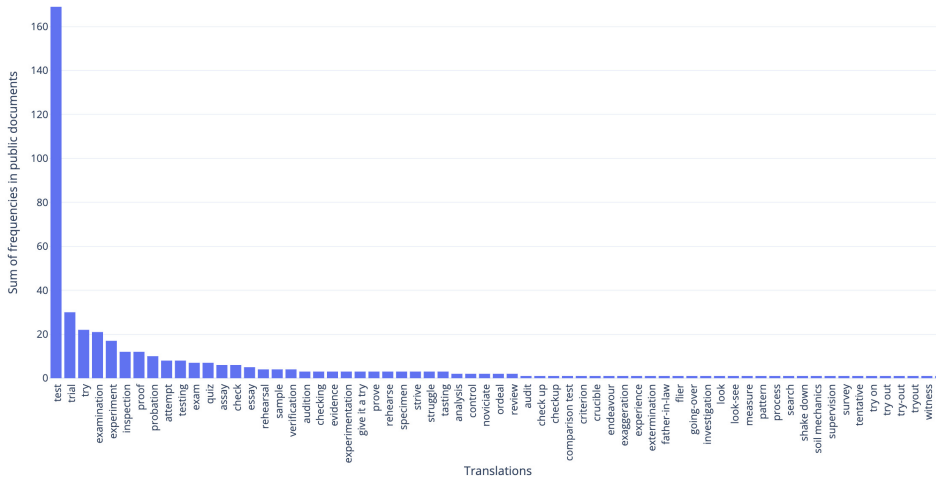

Fig. 1. Sum of reverse translation frequency of the word *test* in public documents of different languages.

A total of 44 languages used non-Latin charset. For these languages, the 2nd approach did not make sense to use. For the remaining languages, the meaning was completely identical in 43 languages and the same or similar meaning in 20 cases. We found only 2 languages (Hungarian[8] and Latvian[9]), in which the word *test* has a completely different meaning, such as *body*, *hew*, or *tool* (nothing related to testing). The analysis shows that the word *test* will actually refer to the testing process in the code and the meaning can vary in very rare cases. Only the word *test* will be searched for in this experiment because of the rate limitations of the GitHub API (explained in section 3.4).

3.3 Java testing frameworks

Usually there is a reason for the developer to use the word *test* in the testing code. In this context, the crucial question is whether developers are motivated to use the word *test* in their code. The developer is greatly influenced by the test framework, which

[8] `https://translate.google.com/?sl=hu&tl=en&text=test`

[9] `https://translate.google.com/?sl=lv&tl=en&text=test`

teaches him or her different habits. As a part of this study, we analyzed 50 Java unit testing frameworks, extensions, and supporting libraries to determine whether the use of the word *test* during test implementation is optional, recommended, or mandatory (see Table 1). The list of testing frameworks was created as a part of this study from different sources, such as blogs, technical reports, research papers, etc.

Framework analysis was performed in early May 2020. Because it is sometimes difficult to find the boundary between unit and integration testing, the table lists frameworks supporting integration testing under the *unit testing* category. Information about the first version and the last commit may be interesting in terms of the framework lifetime and its occurrence in projects. Projects marked as *archived* or *test generators* in Table 1 were excluded from further analysis for the following reasons: 1. archived projects usually had unavailable documentation or were never released; 2. test generators produce tests that are not based on the programmer's mental model, but are generated automatically (semi-randomly), which is not interesting from code comprehension point of view.

It can be seen that 37 of 50 frameworks require the word *test* as method/class annotation (`@Test`) or part of its name (`testMethod`, `methodTest`). This condition is due to the fact that the listed frameworks are mostly extensions that depend on one of the base frameworks, such as *JUnit* or *TestNG*. Different versions of *JUnit* are listed separately, because test labeling differs between them (annotations vs. method name format). A deeper analysis of frameworks' JavaDocs revealed that many frameworks include other classes, methods, or annotations that include the word *test* in their names. Although the use of these methods is not mandatory, it may support the search. At the same time, it should be noted that mostly behavioral frameworks do not use the *test* convention. This is due to the thinking of the developer, whose role in BDD is not to write a test, but a scenario or specification. Such frameworks often use domain specific languages (DSL) to simplify usage for non-programmer team members.

3.4 Searching projects and data gathering

The whole process of data gathering can be seen in Figure 2. GHTorrent provided 140 million of GitHub projects. From this set all deleted, non-Java or duplicated projects have been removed. After cleaning the initial data, a total of 6.7 million projects were kept for further analysis.

GHTorrent contained only basic meta data about the projects, which was not sufficient for our needs. Given the meaning of the word *test* (see section 3.2) it was decided that searching for *test* across all projects would be beneficial. The GitHub API provides a code search[10] endpoint, which index only repositories that are originals (for code search), i.e., not forks. Repository forks are not searchable unless the fork has more stars than the parent repository, therefore, such projects

[10] `https://docs.github.com/en/rest/reference/search`

Table 1. Analyzed unit testing frameworks and extensions for Java.

Name	Package for import	Framework type	First version	Last commit	Must include "test"
SpryTest	N/A	U	N/A	N/A (archived)	N/A
Instinct	N/A	B	24.01.2007	07.03.2010 (archived)	N/A
Java Server-Side Testing framework (JSST)	N/A	U	17.11.2010	17.11.2010 (archived)	■
NUTester	N/A	U	05.02.2009	27.03.2012 (archived)	N/A
SureAssert	N/A	A	29.05.2011	04.02.2019 (archived)	N/A
Tacinga	N/A	U	14.02.2018	22.02.2018 (archived)	N/A
Unitils	N/A	U	29.09.2011 (v3.2)	08.10.2015 (archived)	N/A
Cactus	org.apache.cactus	U	11.2008	05.08.2011 (archived)	■
Concutest	N/A	U	30.04.2009	12.01.2010 (archived)	■
Jtest	N/A	G	1997	21.05.2019 (last release)	■
Randoop	N/A	G	23.08.2010	05.05.2020	■
EvoSuite	N/A	G	25.12.2015 (v1.0.2)	30.04.2020	■
JWalk	N/A	G	19.05.2006	14.06.2017	■
TestNG	org.testng	U	31.07.2010 (v5.13)	11.04.2020	■
Artos	com.artos	U	22.09.2018	19.04.2020	■
JUnit 5	org.junit	U	10.09.2017	02.05.2020	■
JUnit 4	org.junit	U	16.02.2006	10.04.2020	■
JUnit 3	junit.framework	U	N/A	N/A	■
BeanTest	info.novatec.bean-test	U	23.04.2014	02.05.2015	■
GrandTestAuto	org.GrandTestAuto	U	21.11.2009	22.01.2014	■
Arquillian	org.jboss.arquillian	U	10.04.2012	21.04.2020	■
EtlUnit	org.bitbucket .bradleysmithllc.etlunit	U	02.12.2013 (v2.0.25)	04.04.2014	■
HavaRunner	com.github.havarunner	U	16.12.2013	08.06.2017	■
JExample	ch.unibe.jexample	U	2008	N/A	■
Cuppa	org.forgerock.cuppa	U	22.03.2016	01.10.2019	■
DbUnit	org.dbunit	U	27.02.2002	24.02.2020	■
GroboUtils	net.sourceforge.groboutils	U	20.12.2002	05.11.2004	■
JUnitEE	org.junitee	U	23.07.2001 (v1.2)	11.12.2004	■
Needle	de.akquinet.jbosscc.needle	U	N/A	16.11.2016	■
OpenPojo	com.openpojo	U	13.10.2010	20.03.2020	■
Jukito	org.jukito	U/M	25.01.2011	17.04.2017	■
Spring testing	org.springframework.test	M/U	01.10.2002	06.05.2020	■
Concordion	org.concordion	U/SbE	23.11.2014 (v1.4.4)	27.04.2020	□
Jnario	org.jnario	B	23.07.2014		□
Cucumber-JVM	io.cucumber	B	27.03.2012	04.05.2020	□
Spock	spock.lang	B	05.03.2009	01.05.2020	□
JBehave	org.jbehave	B	2003	23.04.2020	□
JGiven	com.tngtech.jgiven	B	05.04.2014	10.04.2020	■
JDave	org.jdave	B	18.02.2008	17.01.2013	□
beanSpec	org.beanSpec	B	15.09.2007	27.06.2014 (alpha)	□
EasyMock	org.easymock.EasyMock	M	2001	10.04.2020	■
JMock	org.jmock	M	10.04.2007	23.04.2020	■
JMockit	org.jmockit	M	20.12.2012	13.04.2020	■
Mockito	org.mockito	M	2008	30.04.2020	■
Mockrunner	com.mockrunner	M	2003	16.03.2020	■
PowerMock	org.powermock	M	28.05.2014 (v1.5.5)	30.03.2020	■
AssertJ	org.assertj	A	26.03.2013	05.05.2020	■
Hamcrest	org.hamcrest	A	01.03.2012	06.05.2020	■
XMLUnit	org.xmlunit	A	03.2003	04.05.2020	■

Legend: U – unit; B – behavioural; A – assert; M – mock; G – generator;
SbE – specification by example

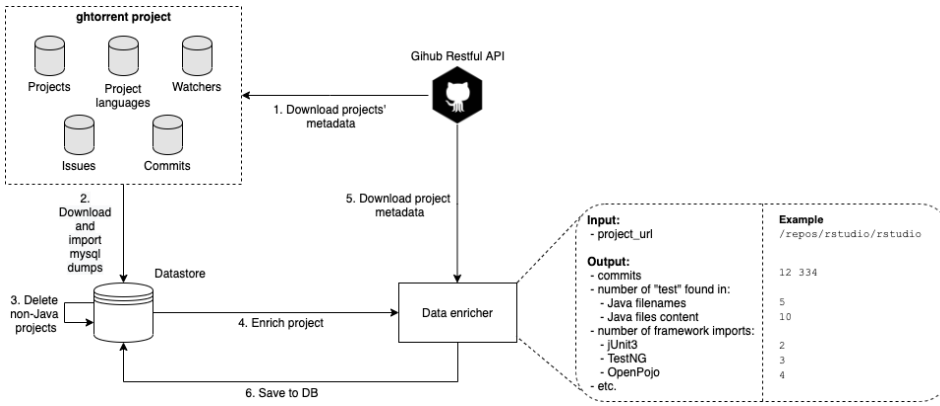

Fig. 2. The GitHub data mining process for the study.

were also removed. If the project has been detected as deleted, private, or blocked by GitHub during querying code search, it has been not considered. Finally, a total of 4.3 million projects were included. The GitHub code search API has the following limitations:

- up to 1,000 results for each search;
- up to 30 requests per minute (authenticated user);
- global requests rate limited at 5,000 requests per hour;
- only files smaller than 384 KB and repositories with fewer than 500,000 files are searchable.

To deal with these limitations, a script was created for data acquisition, which was executed in parallel on 20 instances with different internet protocol (IP) addresses. For each instance, 10 GitHub personal access tokens, which the script cycled, were reserved. The tokens were provided by experiment supporters. Another issue was that our requests were often evaluated as abuse, which significantly slowed down the whole process of data gathering. For each project, two requests to the GitHub code search API were issued, as presented in Table 2. Gathering occurrence of the word *test* took 43 days in total.

Table 2. The GitHub API requests used to search the string *"test"* in a project.

Search *"test"* in	Example request at `https://api.github.com/search/code`
Java files content	`?q=test+in:file+language:java+repo:apache/camel`
Java filenames	`?q=filename:test+language:java+repo:apache/camel`

3.4.1 Code search strategy

GitHub indexes only the default branch code (usually `master`), so the whole analysis was performed only using the default branch. The string *"test"* can also be a part of other words, e.g. *fas**test***, *las**test***, *thisis**test**framework*. There exist 532 such words containing *test*[11] in total. To avoid inaccuracies when searching for a word of the selected string, false positives must be excluded from the search. When using regular GitHub search, the search term will appear in the results when driven by the following rules:

- string uses camel case convention without numbers[12], e.g., `myTest`,
- string uses snake case convention, e.g., `my_test`, `test_123`;
- string includes a delimiter or special character (space, ., #, $, @, etc.), e.g., `test.delimiter`, `@Test`;
- search is case insensitive, e.g. `Test sentence`, `test sentence`.

GitHub considers as Java language file any file with `.java` or `.properties` extensions. The same search rules apply to both search types: file content and filename search. Obviously, according to the above rules, GitHub search automatically filters the results, therefore, unwanted words containing the string *test* do not appear in the results. At the same time, it is necessary to note the side effect, i.e. that neither the words `testing` or `testsAllMethods` will be matched. Therefore, any string alternative that should be found should be included in the search outside the GitHub API. The basic variant of the word will suffice, because best practices use the word *test* to indicate the testing method or class.

3.4.2 Selection of relevant projects

When searching for different testing types, the effort is to go through as many projects as possible. Because GitHub contains millions of repositories, it is a challenge to choose the projects that can be the most instructive, and to filter out irrelevant ones. To make the selection as objective as possible, we planned to use *reaper* tool [24], which can assess a GitHub repository in collaboration with *GHTorrent* using project metadata and code: architecture, community, continuous integration, documentation, history, issues, license and unit testing. By evaluating all these metrics, a particular repository can be tagged as a real software project and thus exclude example projects, forks, irrelevant ones, etc.

The last update of the official *reaper* repository[13] was committed in 2018, many of the libraries it used changed their options, therefore, modification of the project was necessary. Many assessment attributes require project files to be available,

[11] `https://www.thefreedictionary.com/words-containing-test`

[12] Numbers can be used, but they are not considered as individual words, e.g. `123Test` or `test123` will not be found.

[13] `https://github.com/RepoReapers/reaper`

so each project needs to be cloned or downloaded as archive. For large projects, it can be gigabytes of data and the size of the project subsequently affects the length of the analysis. To find out whether *reaper* will be beneficial for our study, a manual analysis of 50 projects was performed and the results were compared with the evaluation of *reaper*. All available evaluation attributes were selected except for unit tests assessment because it was limited to *JUnit* and *TestNG* frameworks. The thresholds and weights of particular attributes were preserved by the developers of the tool because these values were considered empirically confirmed.

Because we want to select a sample of projects from which we would learn the most, projects with the highest number of files containing the word *test* in its body and filename were selected for the comparison. The same attributes as used by the *reaper* were taken into account in the manual evaluation, but the relevance of the project for this study was assessed by an observer. Evaluation of 50 projects using the *reaper* tool took 10 days, with the most time being spent on evaluating the project architecture. Many repositories with the highest *test* presence in file content or filename were actually identified as *Subversion* (SVN) mirrors[14] by manual analysis and because there were multiple copies of the same code (caused by the SVN's branching style), the projects were not relevant, but the *reaper* assessed such projects as suitable. According to this significant issue important projects could be lost by assessing project in an automated manner, so it was concluded that it is more efficient to select projects manually driven by the following rules, inspired by existing research:

- **Priority** was given to projects with the highest number of the word *test* in the project (in file content and filename). According to [27] we can expect presence of tests in popular projects. If it is assumed that the word *test* will be correlated with the number of test cases in the project, large and long maintained projects are expected, which authors consider the best sample for the study.
- **History**, as evidence of sustained evolution. Projects under 50 commits were excluded (inspired by the *reaper*) because they represented small or irrelevant projects. Those projects that contained a large number of commits (more than 1 000 per day), considered committed by a robot, were also excluded.
- **Originality** was evaluated by comparing the `readme` file for similarities in other repositories. By such comparison it is possible to detect clones and similar repositories [34]. Jiang et al. [14] found that developers clone repositories to submit pull requests, fix bugs, add new features, etc. The problem is that developers often do not create forks but project clones (a manual copy of a project), but `readme` file is often unchanged.
- **Community**, as evidence of collaboration, was assessed by number of contributors in the project. The more developers participate in the project, the more likely it is that the (testing) code will be written in a different style.

[14] e.g. `https://github.com/zg/jdk`, `https://github.com/dmatej/Glassfish`, `https://github.com/svn2github/cytoscape`

3.4.3 Searching Java testing frameworks

Inspired by Stefan et al. [30], in order to monitor the impact of frameworks on test writing, we searched projects' code via the GitHub API for imports of any of the testing frameworks from Table 1 (excluding generators and archived projects). This way projects with different frameworks were achieved. Only projects that contained the word *test* in the Java file body at least once were queried. Because there was a large number of requests (37 per single project), the project set was limited to 500,000, ordered by the number of Java files containing the word *test* in its body. The search string was bounded by quotes because GitHub API normally splits words using special characters (e.g. dot) resulting in finding irrelevant results. Example search request: `https://api.github.com/search/code?q="org.testng"+in:file+language:java+repo:apache/camel`.

For each framework we created a separate list of projects, again sorted by the occurrence of the word *test* in the project, to find projects with a high number of test cases if possible. Original repositories of the searched framework were removed from the analysis (e.g. when searching for JUnit, original JUnit framework repository was excluded). Subsequently, the selection of relevant projects was performed according to the steps mentioned in the section 3.4.2. For some frameworks, e.g. *JExample*[15], which were created as a part of the research [19], no software repositories with business focus were found and as a consequence, it was necessary to include also example, homework, or cloned/forked ones, if the original one was not publicly available.

3.5 Repository analysis

Three different data sets were received by searching via GitHub API: 1. the word *test* in filename, 2. the word *test* in file content, 3. frameworks' imports in file content. First four relevant and top projects (highest *test* or framework's import string occurrence) were manually investigated from each set in order to find out the test writing practices. The projects were cloned[16] and to keep the consistency between the *test* search and the manual analysis, the project was reverted to the timestamp of GitHub API download using the following command:

```
git checkout `git rev-list -n 1 --before="<DOWNLOADED_AT>" "<DEFAULT_BRANCH>"`
```

For each selected project all files containing the word *test* or framework's import in file content or filename has been selected as possible option for manual analysis. The project files that contained the largest occurrence of the word *test* and framework's import in their content (expected higher number of tests) were analyzed as first. During the investigation of tests from different authors and from different projects, we created an automated supportive method for detecting the number of

[15] `https://github.com/akuhn/jexample`

[16] `git clone`

test cases in a file. It does not require compiling the code, such as for computing code coverage, or building abstract syntax tree (AST), e.g. indexing in an IDE.

Regardless of the framework, it is advisable to investigate the count of the following attributes of a source file containing the word *test*:

1. *Annotations* `@Test` — very popular mostly thanks to *JUnit* and *TestNG*.
2. *Methods containing* `test` *in the beginning of the name* — best practices leads developers to use this convention (also for historical purposes).
3. *Methods containing* `Test` *in the end of the name* — an alternative of previous one.
4. *Public methods* — possibly all public methods of a test class can be considered as tests.
5. *Occurrence of* `main` — customized testing solutions are executed via `main()`.
6. *File path containing* `test` — should relate to testing.
7. *Classes containing* `$` *in the name* — the character `$` in a class name mostly denotes a generated code[17] that should not be analyzed.
8. *Total number of* `test` *occurrence in file content* — to reveal the relation between executable test cases and the word *test* presence in the content.

All listed metrics (counts of occurrence in a file) were saved for each analyzed file. The pseudocode for collecting mentioned metrics can be seen in Listing 1 (implementation available at GitHub[18]). The presented algorithm is partly the result of the study, because it was created in parallel with the manual analysis. Manual analysis complement the algorithm implementation and vice versa. This algorithm was used to evaluate the test identification for each Java file containing the word *test*. Subsequently, the automated identification was checked during the manual analysis to determine the correct number of test cases and the metric used for the calculation (e.g., the number of annotations and public methods can be the same, but the relevant number of tests can only come from one of them). It is necessary to identify the number of particular test cases in order to link a specific test case with the unit under test (UUT) and it's specific method. Each test case is likely to represent a unique use case and thus unique information to enrich the production code.

Gathered metadata about test case identification were analyzed from different perspectives. As it is not feasible to analyze all tests in a repository, it can be assumed that the testing style in a project is uniform, and rather analyze more projects implemented by different developers. Test classes with the highest number of the following attributes were analyzed: 1. `@Test` annotations, 2. public methods with names starting with `test`, 3. public methods with names ending with `Test`,

[17] `https://docs.oracle.com/javase/specs/jls/se11/html/jls-3.html#jls-3.8`

[18] `https://github.com/madeja/unit-testing-practices-in-java/blob/master/AnalyzeProjectCommand.php`

```
Algorithm predictTests(filePath)
    Input: File path to analyze.
    Output: List of statistical data

    content := load filePath content and remove comments
    nonClassContent := remove all class content, keep only content outside of it
        such as imports or class annotations
    classContent := remove all content outside of the class block and keep only
        first-level methods without body using /\{([^\{\}]++|(?R))*\}/

    annotations := matches count of regex /@Test/ in classContent
    startsWithTest := matches count of regex
        /public +.*void *.* +[Tt]est[a-zA-Z\\d$\_]* *\(/
        in classContent
    endsWithTest := matches count of regex
        /public +.*void *.* +[a-zA-Z$\_]{1}[a-zA-Z\\d$\_]*Test *\(/
        in classContent
    publicMethods := matches count of regex /public +.*void +.*\(/
        in classContent
    includesMain := matches count of /public +static +void +main.*\(/
        in classContent

    hasDollar := if $ in filename, then true, else false
    testInPath := if "/test" in filePath, then true, else false

    if TestNG import found in content, then
        if @Test found in nonClassContent, then
            testCaseCount := publicMethods
        else
            testCaseCount := annotations
    else if JUnit4 import found in content, then
        testCaseCount := annotations
    else if JUnit3 import found in content, then
        testCaseCount := startsWithTest
    else if startsWithTest > 0, then
        testCaseCount := startsWithTest
    else if annotations > 0, then
        testCaseCount := annotations
    else
        testCaseCount := 0

    return annotations, startsWithTest, endsWithTest, publicMethods
            includesMain, hasDollar, testInPath, testCaseCount
```

Listing 1: Pseudocode of the algorithm for gathering metadata and identified number of tests in a Java source file.

4. `main` method, 5. word *test* occurrence. For framework–dependent searches there was an additional analysis of files with the highest framework import occurrence in the content.

4 RESULTS

Using the automated script all repositories' files from Table 3 were processed, 38 repositories and 170,076 classes altogether, from which 803 classes and 20,340 test methods were manually investigated. Some special practices in terms of structure

of the testing code or the developer's reasoning were observed. The first 4 projects from Table 3 represent repositories with the largest occurrences of the word *test* in the filename, another 4 in file content and other repositories represent the top import occurrence of a particular framework.

Table 3. Statistics of the investigated repositories.

Repository	**Framework**	**Analyzed classes**		**Analyzed tests**		**Java KLOC**	T_A
		A	**M**	**A**	**M**		
openjdk/client	testng, junit	30410	130	30410	1661	5149	20798
SpoonLabs/astor	junit	30331	36	30331	1548	2338	13324
apache/camel	junit	10438	81	10438	625	1240	6847
apache/netbeans	testng, junit	13056	78	13056	1627	5009	11908
JetBrains/intellij-community	testng, junit	20375	49	20375	4805	3842	13630
SpoonLabs/astor	testng, junit	30331	44	30331	5883	2338	13324
corretto/corretto-8	testng, junit	13688	10	13688	1659	3638	10792
aws/aws-sdk-java	junit	28574	18	28574	302	3680	20528
wildfly/wildfly	arquillian	5109	24	5109	123	548	3553
eclipse-ee4j/cdi-tck	arquillian	4758	30	4758	139	97	2748
resteasy/Resteasy	arquillian	2821	13	2821	144	220	1675
keycloak/keycloak	arquillian	1681	16	1681	104	396	1286
jsfunit/jsfunit	cactus	222	13	222	125	21	142
bleathem/mojarra	cactus	737	16	737	250	171	556
topcoder-platform /tc-website-master	cactus	1635	8	1635	42	366	1199
apache/hadoop-hdfs	cactus	325	4	325	20	101	282
zanata/zanata-platform	dbunit	770	21	770	171	197	554
B3Partners/brmo	dbunit	145	18	145	37	47	106
gilbertoca/construtor	dbunit	145	18	145	64	24	53
sculptor/sculptor	dbunit	153	11	153	101	26	103
geotools/geotools	groboutils	3424	5	3424	5	1272	3659
notoriousre-i-d/ce-packager	groboutils	107	11	107	75	46	91
tliron/prudence	groboutils	16	2	16	3	13	11
MichaelKohler/P2	jexample	36	12	36	53	4	24
akuhn/codemap	jexample	132	15	132	286	41	112
wprogLK/TowerDefenceANTS	jexample	17	3	17	50	9	12
rbhamra/Jboss-Files	needle	44	21	44	30	5	30
akquinet/mobile-blog	needle	19	10	19	33	2	10
s-case/s-case	needle	46	15	46	13	39	33
dbarton-uk/population-pie	needle	7	6	7	16	1	4
abarhub/rss	openpojo	26	2	26	3	6	20
BRUCELLA2 /Prescriptions-Scolaires	openpojo	25	19	25	40	10	18
jpmorganchase/tessera	openpojo	382	8	382	12	45	234
tensorics/tensorics-core	openpojo	161	3	161	1	24	85
orange-cloudfoundry /static-creds-broker	jgiven	21	11	21	33	2	16
eclipse/sw360	jgiven	175	4	175	51	56	161
Orchaldir /FantasyWorldSimulation	jgiven	54	13	54	198	7	37
kodokojo/docker-image-manager	jgiven	11	5	11	8	3	8
SUM		**170076**	**803**	**363730**	**20340**	**31033**	**127973**

Legend: A – processed automated; M – investigated manually; KLOC – kilo of lines of code;
T_A – average time of automated test case detection in ms.

4.1 Correlation between the word *test* occurrence and test cases

To evaluate the precision of the algorithm from Listing 1, results were compared to manual test identification of 20,340 test cases across all three datasets (the word *test* search in filename, file content and framework import search in file content, see section 3.5). Accuracy of 95.72% for test cases detection was achieved by automated identification. Most of false positives and false negatives occurrences were caused by customized testing solutions, e.g. when tests were performed directly from the `main()` function by calling methods of the class. If the naming conventions of the called (testing) methods were not governed by the principles of frameworks (e.g. prepending method name with *"test"* or using public methods), not all test cases were detected in an automated way.

The proposed algorithm was used to identify all tests in all Java classes of projects from Table 3. The script was used for all Java files that contained string *"test"* in the file content or in the filename (in total 170,076 files). Figure 3 shows the correlation with the linear regression line of the word "test" and the number of test cases in particular class. A standard (pearson) correlation coefficient of $r = 0.655$ was reached, that is not strongly significant when considering significance level $\alpha = 0.05$. However, from the perspective of finding projects containing tests, this technique is beneficial and can help future experimenters to filter projects containing tests much more faster. Because projects have different numbers of test classes and use different frameworks, the detailed ratio of the word "test" occurence and test case presence per project can be found at GitHub[19].

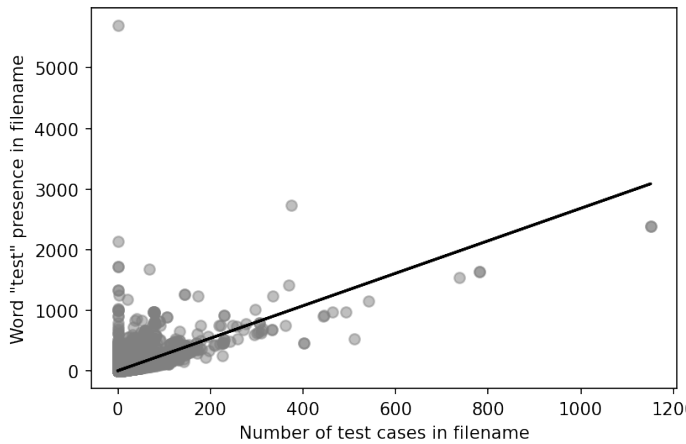

Fig. 3. Correlation of the word "test" presence and number of test cases for analyzed classes by automated script.

[19] `https://github.com/madeja/unit-testing-practices-in-java/blob/master/correlation-boxplot.png`

Due to existing research [20] that identified test files using searching "test" in the file path, when limiting our results to files containing "test" in the path (120,907 files) the correlation coefficient of $r = 0.6649$ was reached. On the other hand, 49,169 classes with 3,855 test cases were discarded. Limiting results to files containing "test" in filename (74,530 files), we reached correlation coefficient $r = 0.7004$ with loss of 95,546 classes and 17,440 test cases. By any limitation the correlation did not significantly changed, therefore, to find as much test cases as possible it is convenient to search for the word "test" in file content. To identify exact number of test cases with their exact location in the project, the script mentioned in section 3.5 can be used.

Occurrence of the function `main` without the 3rd party testing framework (more explained in section 4.3.2) was detected in 26,205 (15.41%) classes containing the word *test* in their content. The proposed algorithm in section 3.5 successfully identified test cases in only 6% classes of this set. Because `main` tests make up a fairly large proportion and the identification of test cases is not clear, it is necessary to investigate this testing style deeper in the future.

H1: There is a correlation between the occurrence of the word *"test"* in the file content and the number of test cases.

The H1 has been refused, because reached correlation coefficient $r = 0.655$ was weakly significant. Filtering data by searching the word "test" in file path or filename did not changed correlation significantly but a large number of test cases were lost. Nevertheless, searching for string "test" in project files' content may help future experiments to fastly estimate the number of test cases in a project. The exact identification and mostly the location of the test cases in the project can be evaluated as the next step, e.g. using the proposed script with the achieved accuracy of 95.72%.

4.2 Efficiency of the proposed automated test case identification

Executing a full code analysis, e.g. in an IDE, of large project with thousands of kilo of lines of code (KLOC), is a time consuming task. Such example is the project `openjdk/client` from Table 3. To get a faster feedback about tests in a project, the proposed algorithm was used for static source code analysis. Because proposed automated algorithm should run as a part of an integrated development environment (IDE) extension in the future it should be fast enough. To emulate similar environment that a developer can use, a laptop with *2.3 GHz Dual-Core Intel Core i5* CPU and *8 GB 2133 MHz LPDDR3 RAM* was used. In the table 3 can be seen the average time (T_A) of automated analysis executed 10 times. The average time of execution was 158ms per KLOC, which authors consider as a satisfactory response time in terms of user experience for use in an IDE extension.

4.3 Revealed testing practices

In related work (section 2) there are best practices and recommendations that developers can follow and therefore can be expected in the code. During the manual investigation of multiple repositories containing tests, we identified special testing practices used by developers, which are described in the following paragraphs. The listings that are given as examples come from the analyzed repositories, but the code was simplified for presentation purposes. Code listings refer to GitHub[20] repository of this paper.

4.3.1 Testing using 3rd party frameworks

Regular test. Tests that follow best practices and avoid test smells fall into this category. They represent the majority of occurrences in the projects and since these approaches are already described in the available literature [25, 21, 9], this group will not be given a detailed attention. However, the basic aspect of such tests is that information about context and evaluation are available directly in the particular test method (considering also test setup, teardown and fixtures), thanks to which the test comprehension is straightforward.

Master test. This testing code style represents test classes which contain only one executable test method (see GitHub[21]). *JUnit* will consider only the `all()` method as a test case, because it is annotated with `@Test` annotation. Other methods are considered auxiliary ones. The problem with such a notation is the complexity of test comprehension. If the test fails, the developer only has information that the test `all` failed, but does not know what the test should have verified, what data was used, etc.

According to the best practices, it should be clear from the test name what the test verifies. In this context, from a semantic point of view, it is possible to consider methods as test cases on lines 1-8. The mentioned methods are crucial in terms of failure and understanding of the test, and from the method name it is also clear what the test verifies. Another disadvantage of these test types is the *assertion roulette* test smell [32], because iterations of the test over the input data make it difficult to determine which data caused the test failure and whether the input data do not interfere with each other between the tests.

Reverse proxy test. If a separate test is written for each use case, the recommendations are met, but this does not mean that it will be easy to understand. There are tests that call one auxiliary method in multiple tests and the result is evaluated in the auxiliary method. According to the test evaluation manner, they can be divided into:

[20] `https://github.com/madeja/unit-testing-practices-in-java`

[21] `https://github.com/madeja/unit-testing-practices-in-java/blob/master/examples/c_masterTest.java`

1. *Result evaluation via method name* (see GitHub[22]).
2. *Result evaluation via internal object state* (see GitHub[23]).

The 1st approach is much more difficult to comprehend due to the high degree of abstraction. It is not clear directly from the test method code (L5-7) what is compared during the test, because the input data are loaded from a file determined by the test method name (L2). In the `JetBrains/intellij-community` project, from which the example is given, the `doTest()` method is the general one and it was necessary to investigate multiple classes to comprehend how tests are evaluated. At the same time, too generic auxiliary method can result in the *general fixture* test smell.

The 2nd approach is similar to the previous one, but uses the internal state of an object that is initialized before a particular test during test setup or the `enum` type with different method implementations. The problem of these methods might arise if the method accepts an input parameter, which is later used to change the control flow. If the same test is called with different input data, the test logic does not change and therefore it is the same test. However, if the control flow changes in the test, e.g. by some variable value, it can be considered as a separate test (different flow, different test). If the same auxiliary method is called more than once, there may be 2 different tests, which contradicts best practices and makes the comprehension difficult.

Multiple test execution. Mostly server-side application test different use cases, which require an action after the execution of base functionality, e.g. whether the right content is shown after main test execution (see GitHub[24]). Because in the example *JUnit3* is used, every public method prepended by *test* is considered as test case, so `testEcho()` is executed twice; as a single test case and as a part of `testA4JRedirect()`.

4.3.2 Customized testing solutions

Custom testing practices of testing practices are classic Java programs executable via `main()` function, whose task is to verify the functionality of the production code. Such tests are often written due to the possibility of configuring the execution via command line parameters, which allows variability of test execution. On the other hand, tests should not be so environmentally dependent that they need to be configured to such an extent. The second possibility for why to write such tests is that they clarify the testing code with a large number of cases. Test methods are called directly from `main()` and, if necessary, also the environment setup is

[22] `https://github.com/madeja/unit-testing-practices-in-java/blob/master/examples/c_reverseProxyMethod.java`

[23] `https://github.com/madeja/unit-testing-practices-in-java/blob/master/examples/c_reverseProxyObject.java`

[24] `https://github.com/madeja/unit-testing-practices-in-java/blob/master/examples/c_multipleExecution.java`

performed in this function. The following ways of calling test methods and objects were observed:

- *Calling methods one by one:* all testing methods are manually called from `main()` together with parameters.
- *Calling methods according to input data:* by iterating the test data, specific tests are called based on the current data.
- *Helper function that returns an array of test cases:* the helper method returns an array of instances created from abstract classes, whereas the abstract methods (which represent test cases) are implemented during the instance creation. The `main()` contains an iteration over the array of object instances.
- *Iterating values of* `enum`*:* similar to the previous one, but it iterates over `enum` values. When creating the `enum`, the method of test class is implemented and the data is set. The test class has its own implementation of a method and state in each iteration.
- *Calling constructor:* in the `main` function the testing class instance is created and the tests are called from the constructor.

There is a problem of how to identify such test using automated way and how to determine number of tests in such a class. The `main()` function also occurs in classic tests (e.g. to run test outside of IDE or without a build automation tool[25]), e.g. based on *JUnit* or *TestNG*. The function can also be found in modified runners of testing frameworks. To clearly distinguish the presence of a customized solution without any framework, it is possible to check the presence of the framework import — if a class contains the `main()` function and an import together, it is a runner or regular test based on the framework, not a customized solution.

Other interesting ways of writing customized tests were also observed. For example, in the `openjdk/client` repository, there were tests for trichotomous relations for which a custom `@Test` annotation was implemented (see GitHub[26]). The annotation is used to indicate the test and, at the same time, to define the type of comparison in the method (L1, L4). Thanks to the word *test* usage, it is possible to detect the correct number of tests, in similar way as for *JUnit*. In this example, the impact of 3rd party framework on the developer's customized solution is visible. There are many tests in the repository using standardized frameworks, therefore the usage of `@Test` annotation is a logical way of defining a test case. The example shows the execution of every annotated method 20,000 times (L9). Writing tests manually using a framework would not be as effective and would be difficult to comprehend. On the other hand, such tests in large iterations can easily give rise to the *assertion roulette* test smell, which makes it difficult to identify a test failure.

[25] https://junit.org/junit4/faq.html

[26] `https://github.com/madeja/unit-testing-practices-in-java/blob/master/examples/c_main1.java`

While in the previous case the test was evaluated using asserts, there are also approaches that have their own error handling. E.g. in the same repository for all *ResourceBundle* classes, a helper test class `RBTestFmwk` has been implemented, which represents a custom framework and test classes inherit from it. The framework provides the processing of the `main()` function parameters, performing tests, and processing results. The test methods to be performed are defined as input parameters. The disadvantage is that when performing such tests, it is necessary to know the internal structure of the class, at least method names that need to be performed.

In general, the following risks were observed by analyzing other `main` testing methods:

- *Execution interruption* — If a test fails, execution may be completely interrupted and no further tests will be performed (e.g. raised exception).
- *Failure identification* — Because testing is often performed repeatedly over different data, it can be difficult to identify the exact cause of test failure and in some cases may require debugging the test code.
- *Dependence* — Tests often use the same sources or data for testing and may affect the results of other tests. Also, the tests are often order-dependent and the test order randomness was not found in any repository.

As mentioned in section 4.1, because of the high diversity of writing such tests, it is necessary to carry out an extensive study dealing solely with this issue, to find a way to precisely identify such test cases.

H2: Tests are usually constituted by means of test frameworks, but there are also other ways of automated code testing.

The H2 has been confirmed. Third party frameworks are used mostly for testing (84.59%), but there are also customized testing solutions in the form of classic Java programs, whose task is to test production code. Such customized solutions are mainly used to perform a large number of tests, which are performed multiple times and this way the implementation is simplified.

5 THREATS TO VALIDITY

Internal validity: The study relied on GHTorrent databank and GitHub API search algorithm to identify relevant projects. Because only projects with a majority of Java language were selected, testing practices in projects, where Java was not a major language could have been lost. Test classes that did not use the word *test* to indicate a test case were also lost. Searching for test cases was based on best practices and rules of the identified frameworks, but there may still exist other ways of how to identify them. Manual classification was based on observers' experiences

and identification of practices out of the generally known recommendations (best practices, test smells, etc.).

Test case detection results were compared to manual ones with the accuracy of 95.72%. As stated, it is necessary to further investigate customized testing solutions that use regular Java programs to test the production code. The implementation of such programs is often diametrically different and it is difficult to identify test cases. Real test cases were identified by the script in 6% of classes containing `main()` function.

External validity: To provide generalizable results, 20k of test cases were analyzed manually and 170k by an automated way. Also, the meaning and occurrence of the word *test* was analyzed for different natural languages and test frameworks. The results can be used to identify test cases in Java-based projects or projects with a different programming language with the usage of similar testing conventions. Despite the presented observations, our findings, as is usual in empirical software engineering, may not be directly generalized to other systems, particularly to commercial or to the ones implemented in other programming languages.

6 CONCLUSION AND FUTURE WORK

This paper presented a large empirical study of Java open source GitHub projects to better understand how to identify test cases and and their exact location in the project without the need of deep and time-consuming dynamic code analysis. An algorithm based on searching the word *test* in the repository files content or filenames was proposed and, at the same time, the unusual testing practices were investigated. In total 20,340 test cases in 803 classes were investigated manually and 170k classes by an automated way. We summarise the most interesting findings from our study:

- There is not strong correlation between the word *test* occurrence and test cases presence in a class.
- Searching for the word *test* in the file content can be used to identify projects containing test.
- Using static file analysis, the proposed algorithm is able to correctly detect 95% of test cases.
- Approximately 15% of the analyzed files contains *test* in the content together with `main()` function whose represent regular Java programs that test the production code without using any third-party framework. Success rate of identification of such test cases is very low because of implementation diversity.

Several test writing styles were found and classified, along with code samples of the analyzed repositories. Possible origins of test smells and other code comprehension defects were also mentioned. Based on these findings the following main contribution of this paper are concluded:

- Possibility of fast and automated test case identification together with the exact location in the project.

- Finding of correlation coefficient $r = 0.655$ between the occurrence of the word *test* and test case in a file, which allows to threshold projects or files for similar analysis.
- Overview of observed testing practices with respect to existence of customized testing solutions with emphasis on places in testing code usable for mining information about the production code.

As future work, we plan to find a solution for an accurate identification of test cases in customized solutions. We believe that by studying testing practices, it will be possible to train artificial intelligence to automatically recognize tests by the structure and nature of the code. At the same time, we would like to focus on mining tests for information that could support the production source code comprehension and streamline the development process.

7 ACKNOWLEDGEMENT

This work was supported by project VEGA No. 1/0762/19: Interactive pattern-driven language development.

REFERENCES

[1] Beller, M., Gousios, G., Panichella, A., and Zaidman, A. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, Association for Computing Machinery, p. 179–190.

[2] Bissi, W., Serra Seca Neto, A. G., and Emer, M. C. F. P. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology 74* (2016), 45 – 54.

[3] Butler, S., Wermelinger, M., and Yu, Y. Investigating naming convention adherence in java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2015), pp. 41–50.

[4] Corritore, C. L., and Wiedenbeck, S. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies 50*, 1 (1999), 61 – 83.

[5] Cruz, L., Abreu, R., and Lo, D. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering 24*, 4 (2019), 2438–2468.

[6] Demeyer, S., Ducasse, S., and Nierstrasz, O. *Object-oriented reengineering patterns*. Elsevier, 2002.

[7] Ellims, M., Bridges, J., and Ince, D. C. Unit testing in practice. In *15th International Symposium on Software Reliability Engineering* (2004), pp. 3–13.

[8] Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., and Juristo, N. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering 43*, 7 (2017), 597–614.

[9] Garcia, B. *Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. Packt Publishing Ltd, 2017.

[10] Gopinath, R., Jensen, C., and Groce, A. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering* (2014), pp. 189–200.

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

[12] Hemmati, H. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security* (2015), pp. 151–156.

[13] Hilton, M., Bell, J., and Marinov, D. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), ASE 2018, Association for Computing Machinery, p. 53–63.

[14] Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P. S., and Zhang, L. Why and how developers fork what from whom in github. *Empirical Software Engineering 22*, 1 (2017), 547–578.

[15] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, Association for Computing Machinery, p. 654–665.

[16] Kochhar, P. S., Lo, D., Lawall, J., and Nagappan, N. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability 66*, 4 (2017), 1213–1228.

[17] Kochhar, P. S., Thung, F., and Lo, D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015), pp. 560–564.

[18] Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., and Lo, D. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), pp. 1–10.

[19] Kuhn, A., Van Rompaey, B., Haensenberger, L., Nierstrasz, O., Demeyer, S., Gaelli, M., and Van Leemput, K. Jexample: Exploiting dependencies between tests to improve defect localization. In *Agile Processes in Software Engineering and Extreme Programming* (Berlin, Heidelberg, 2008), P. Abrahamsson, R. Baskerville, K. Conboy, B. Fitzgerald, L. Morgan, and X. Wang, Eds., Springer Berlin Heidelberg, pp. 73–82.

[20] Leitner, P., and Bezemer, C.-P. An exploratory study of the state of practice

of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (New York, NY, USA, 2017), ICPE '17, Association for Computing Machinery, p. 373–384.

[21] Lewis, W. E. *Software testing and continuous quality improvement*. CRC press, 2017.

[22] Linares-Vásquez, M., Bernal-Cardenas, C., Moran, K., and Poshyvanyk, D. How do developers test android applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), pp. 613–622.

[23] Madeja, M., and Porubän, J. Tracing Naming Semantics in Unit Tests of Popular Github Android Projects. In *8th Symposium on Languages, Applications and Technologies (SLATE 2019)* (Dagstuhl, Germany, 2019), R. Rodrigues, J. Janousek, L. Ferreira, L. Coheur, F. Batista, and H. G. Oliveira, Eds., vol. 74 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 3:1–3:13.

[24] Munaiah, N., Kroh, S., Cabrey, C., and Nagappan, M. Curating github for engineered software projects. *Empirical Software Engineering 22*, 6 (2017), 3219–3253.

[25] Nayyar, A. *Instant Approach to Software Testing: Principles, Applications, Techniques, and Practices*. BPB Publications, 2019.

[26] Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. On the distribution of test smells in open source android applications: an exploratory study. In *CASCON* (2019), pp. 193–202.

[27] Pham, R., Singer, L., Liskin, O., Filho, F. F., and Schneider, K. Creating a shared understanding of testing culture on a social coding site. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 112–121.

[28] Scalabrino, S., Linares-Vásquez, M., Poshyvanyk, D., and Oliveto, R. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (2016), pp. 1–10.

[29] Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2018), pp. 1–12.

[30] Stefan, P., Horky, V., Bulej, L., and Tuma, P. Unit testing performance in java projects: Are we there yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (New York, NY, USA, 2017), ICPE '17, Association for Computing Machinery, p. 401–412.

[31] Sulír, M., Bačíková, M., Madeja, M., Chodarev, S., and Juhár, J. Large-scale dataset of local java software build results. *Data 5*, 3 (2020), 86.

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

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

[34] Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q., and Sun, J. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2017), pp. 13–23.

Matej Madeja was born in 1992 in Kežmarok, Slovakia. In 2017 he graduated (MSc) at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University of Košice. He defended his master's thesis in the field of Informatics. Currently, he is a PhD student in the same department. His research is focused on improvement of program comprehension efficiency, source code testing techniques and teaching of programming.

Jaroslav Porubän is a Professor and the Head of Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is a member of the Department of Computers and Informatics at Technical University of Košice. Currently the main subject of his research is the computer language engineering concentrating on design and implementation of domain specific languages and computer language composition and evolution.

Michaela Bačíková is an assistant professor and the Head of the Information Systems Laboratory at the Department of Computers and Informatics, Technical University of Košice, Slovakia. She received her Ph.D. in Computer Science in 2014. Since 2014 she is a member of the Department of Computers and Informatics at Technical University of Košice. Currently the main subject of her research is UX, HCI and usability while focusing on the domain-related terminology in user interfaces (domain usability). She also focuses on software languages and innovations in the teaching process.

Matúš Sulír is an assistant professor at the Department of Computers and Informatics, Technical University of Košice, Slovakia. At the same university, he graduated with a Master's degree in Computer Science in 2014 and PhD in 2018. His research is focused on program comprehension, particularly on the integration of run-time information with source code, attribute-oriented programming, and debugging. He is also interested in empirical studies in software engineering.

Jan Juhár is a researcher at the Department of Computers and Informatics, Technical University of Košice. He received his PhD. in Computer Science in 2018. Since 2018 he is a member of Department of Computers and Informatics at Technical University of Košice. His research focuses on program comprehension, programming tools, source code metadata and program projections.

Sergej Chodarev is an assistant professor at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his Master's degree in computer science in 2009 and his PhD degree in computer science in 2012. The subject of his research includes domain-specific languages, metaprogramming and software engineering.

Filip Gurbáľ is a Ph.D. student at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his Ing. in Computer Science in 2020. He is a member of the Computer Network Laboratory at Technical University of Košice. The subject of his research is improving program comprehension using methods and tools. He also focuses on software testing methods and tools.