The Journal of Systems and Software 159 (2020) 110421

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

# Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects

Fabian Trautsch*, Steffen Herbold, Jens Grabowski

*Institute of Computer Science, University of Goettingen, Goldschmidtstraße 7, Niedersachsen, 370077, Germany*

ARTICLE INFO

*Article history:*
Received 16 May 2019
Revised 8 August 2019
Accepted 16 September 2019
Available online 20 September 2019

*Keywords:*
Software testing
Unit testing
Integration testing
Empirical software engineering

ABSTRACT

*Context:* Unit and integration testing are popular testing techniques. However, while the software development context evolved over time, the definitions remained unchanged. There is no empirical evidence, if these commonly used definitions still fit to modern software development.
*Objective:* We analyze, if the existing standard definitions of unit and integration tests are still valid in modern software development contexts. Hence, we analyze if unit and integration tests detect different types of defects, as expected from the standard literature.
*Method:* We classify 38,782 test cases into unit and integration tests according to the definition of the IEEE and use mutation testing to assess their defect detection capabilities. All integrated mutations are classified into five different defect types. Afterwards, we evaluate if there are any statistically significant differences in the results between unit and integration tests.
*Results:* We could not find any evidence that one test type is more capable of detecting certain defect types than the other one. Our results suggest that the currently used definitions do not fit modern software development contexts.
*Conclusions:* This finding implies that we need to reconsider the definitions of unit and integration tests and suggest that the current property-based definitions may be exchanged with usage-based definitions.

## 1. Introduction

Unit testing as well as integration testing are two popular testing techniques that are used in industry and academia. There exist a lot of scientific literature, which is focused on unit testing (Van Rompaey et al., 2007; Fraser and Zeller, 2011; Gambi et al., 2017) or integration testing (Lachmann et al., 2016; Xu et al., 2016; Holling et al., 2016). Furthermore, there exist several development models in which both practices are integrated, e.g., the V-Model Boehm (1984) or the waterfall model Royce (1999).

The definitions for unit and integration tests, like the ones of the Institute of Electrical and Electronics Engineers (IEEE) IEEE (2010) or International Software Testing Qualification Board (ISTQB) International Software Testing Qualification Board, are decades old. However, the software development context has evolved over time, while these definitions remained unchanged. For example, major contributions in the field of software testing that pushed this evolution were the development of xUnit frameworks (e.g., JUnit Team (2017)), which are nowadays used to execute all types of tests including unit, integration, and system tests, or continuous integration systems (e.g., Travis CI GmbH (2017); Jenkins Contributors (2018)). Those frameworks and systems changed the way in which developers test their software. This is highlighted by a proposal that is spread within the development community Dodds (2017). The idea is to change a software testing paradigm: instead of testing a software thoroughly on unit level with a few integration tests, developers propose to test the software mostly on integration level with only a few unit tests. The reasons for this are manifold like, e.g., that integration tests are more realistic Dodds (2017) or more effective Coplien (2014a,b) than unit tests. In addition, unit testing might not be feasible for legacy systems, as the overhead to refactor the whole code base to establish a unit-level testability might be very high (Garousi and Yildirim, 2018). On the other hand, there are also concepts where unit tests have an essential part, e.g., the test pyramid proposed by Cohn (2010) or the 4-testing quadrants (or agile testing quadrants) (Crispin and Gregory, 2009). Some of these concepts where also further studied, e.g., by Google (Wacker, 2015). Within their post they suggest that you should have 70% unit, 20% integration, and 10% end-to-end tests, highlighting the importance of unit tests.

* Corresponding author.
*E-mail addresses:* trautsch@cs.uni-goettingen.de (F. Trautsch), herbold@cs.uni-goettingen.de (S. Herbold), grabowski@cs.uni-goettingen.de (J. Grabowski).

https://doi.org/10.1016/j.jss.2019.110421
0164-1212/© 2019 Elsevier Inc. All rights reserved.

Hence, we can clearly see an evolution of the development context, but the definitions were not adapted over the time. Therefore, we hypothesize that the current definitions, that are used in current textbooks and software testing certifications (e.g., ISTQB certified tester International Software Testing Qualification Board), do not fit to modern software development contexts. Our prior work Trautsch and Grabowski (2017) presents us indications that support this hypothesis. We have shown that developers do not classify their tests according to the definitions of the IEEE or ISTQB. While this shows that the definitions are not used in practice, it does not gives us any indications regarding the merit of the current definitions. Since the main purpose of testing is to reveal defects, we focus on comparing the defect detection capabilities of unit and integration tests in this paper. Software testing textbooks state Spillner et al. (2014); Ammann and Offutt (2016); Myers et al. (2011), that there should be a difference between unit and integration tests in terms of types of defects that they detect. Hence, we evaluate if unit or integration tests are more capable in detecting certain program defect types. If we do not find a difference, we would have another indication that the current definitions of unit and integration tests do not fit the modern software development context.

To steer our research, we defined one central Research Question (RQ):

- **RQ:** Are unit or integration tests more capable in detecting certain program defect types?

To answer our RQ, we have a look at the defect detection capabilities of unit and integration tests in terms of their defect detection capabilities. For this, we first classify tests into unit and integration tests and afterwards collect data about their defect detection capabilities using mutation testing. Then, we perform an in-depth analysis by assessing which types of defects are found by the two different types of testing and if there are any differences in their defect detection capabilities of these defect types. The contributions of this paper can be summarized as follows:

- an in-depth analysis of the defect detection capabilities of unit and integration tests.
- an in-depth analysis of which defect types are found by unit and integration tests.
- a data set, which includes detailed information about the defect detection capabilities of 38,782 test cases for 17 projects.
- a software test analysis framework, which can be used for further research and enable other researchers to contribute to the body of knowledge of empirical software testing research.

The remainder of this paper is structured as follows. In Section 2 we lay the foundations for this paper and discuss related work. Section 3 describes our research methodology, including the approach for our research question, our data collection and analysis procedures, our results, and the replication kit of this paper. We then discuss the results in Section 4. Afterwards, in Section 5, we discuss potential new definitions of unit and integration tests. Section 6 presents the threats to validity to our study, including construct, internal and external threats. Finally, we conclude our paper and describe future work in Section 7.

## 2. Foundations and related work

In this section, we present the terminology that is used along this paper together with a summary of the literature that is related to our research.

### 2.1. Definitions

The IEEE standard ISO/IEC/IEEE 24765-2010 (IEEE, 2010) defines the most important vocabulary for the software engineering world. In the following, we present the definitions that are important for this paper based on this standard.

**Definition 1** (Unit)**.** 1. a separately testable element specified in the design of a computer software component. 2. a logically separable part of a computer program. 3. a software component that is not subdivided into other components. [... ]. (IEEE, 2010)

**Definition 2** (Unit Test)**.** [... ] 3. test of individual hardware or software units or groups of related units (IEEE, 2010)

**Definition 3** (Integration Test)**.** 1. the progressive linking and testing of programs or modules in order to ensure their proper functioning in the complete system. (IEEE, 2010).

### 2.2. Test type classification

There are several approaches present in the literature that try to classify tests into different classes. In the short paper by Orellana et al. (2017), the classification is based on the usage of Maven plugins in project builds. If test classes are executed by the Maven SureFire plugin (Apache Software Foundation, 2017c) they are classified as unit tests, but if they are executed by the Maven FailSafe plugin (Apache Software Foundation, 2017a) they are classified as integration tests. Hence, they reuse the classification of the developers. This can be problematic, as our earlier work (Trautsch and Grabowski, 2017) highlights, that the classification into unit and integration tests, as done by the developers, is not always in line with the definitions. Furthermore, this kind of classification does not allow a fine-grained analysis on the method level. Another problem is, that not all projects differentiate their tests based on the executed Maven plugins. For example, none of the projects that we have used in our case study use the Maven FailSafe plugin, while some tests of the projects are indeed integration tests.

Another approach that classifies tests into different types was proposed by Kanstrén (2008). Kanstrèn proposes a dynamic approach that uses aspect oriented programming to calculate the test granularity of each test, which refers "to the number of units of production code included in a test case... "(Kanstrén, 2008). To achieve this, his approach calculates the number of methods that are covered by each test. Afterwards, the results can be summarized to determine, if, e.g., a system was tested only on low-level (i.e., many tests that executed a small amount of methods) or only on high-level.

In our earlier work (Trautsch and Grabowski, 2017) we determined how many tests are unit tests, according to the definitions of the ISTQB and IEEE for 10 Python projects. We proposed a static analysis approach in which we assessed the number of imported modules in a test to find unit tests. Furthermore, we compared the intention that the developers had (i.e., if they wanted the test to be a unit test) with the outcome of the classification analysis.

We improved our test classification approach in several ways. We switched from a static analysis approach to a dynamic one to resolve the drawbacks of our former approach, i.e., we can now directly determine if a unit is used (and not only imported) in a test case and the approach is robust against the usage of mocking frameworks inside a project. Our new approach is using test coverage data to classify tests into unit and integration tests, which was not possible before. Hence, our approach only needs the coverage data of a test execution run in contrast to the work of Orellana et al. (2017). Furthermore, our approach makes a clear separation between unit and integration tests based on the definition of the IEEE (IEEE, 2010). This is in contrast to the work of Kanstrén (2008), which only provides a general view on the granularity of the tests. Furthermore, our analysis is more fine-grained than before. Instead of classifying a whole test class/module we are able to classify each method separately.

### *2.3. Assessment of defect detection capabilities via mutation analysis*

The assessment of the defect detection capabilities of a set of test cases is done by checking how many defects the test suite can find. There exist several approaches in the literature that can be used. Recently, (Orellana et al., 2017) assessed if unit tests expose more defects than integration tests by using the Travis Torrent dataset (Beller et al., 2017). This data set contains information about project builds, including which test cases failed during the build. The authors of the paper used this information to determine the number defects that were exposed. The assumption in this kind of analysis is that every test that failed during the build of a project exposed a defect. The problem with this approach is that tests are often executed before committing changes to a version control system. Hence, all defects that might be found *before* committing the changes are not recorded. Moreover, this kind of technique is focused on defects that broke the build and therefore the number and type of assessed defects is very limited. First, because tests in a continuous integration system could also fail due to wrong commit behavior (e.g., the developer forgot to commit a certain change to a class). Second, with this technique we could only assess defects that are detected by the continuous integration system. Defects that might get fixed before committing to the continuous integration system are missing. Hence, to answer our RQ we require a controlled environment, where we are able to exclude as many outside influences as possible that might have an influence on the results. We achieve this through mutation analysis.

Mutation analysis provides a systematic way for introducing defects into a software (Papadakis et al., 2017). In mutation analysis we integrate mutants (defects) into the program and check if these are found by test cases. If a test case detects a mutant, i.e. the test case fails, we call the mutant *killed*. Mutants that survive the execution of the test case are called *live* (Papadakis et al., 2017). The *mutation score* is defined as a ratio of the number of killed mutants to the total number of mutants. Unfortunately, some of the generated mutants produce a program which behaves equivalent to the original one (Papadakis et al., 2015; Schuler and Zeller, 2013) and can therefore not be killed. These mutants are called *equivalent*. However, the detection of such mutants is an instance of an undecidable problem (Ammann and Offutt, 2016). Mutation testing can be very time consuming, as a large number of defect-seeded programs must be tested.

The use of mutation analysis for the assessment of the defect detection capabilities of tests, has the underlying assumption that the mutants can construct program failures that are similar to the ones that are created through real defects. There are several studies that provide indications that this is the case. One of the first studies that investigated the relationship between mutants and real defects was done by Daran and Thévenod-Fosse (1996). They found that injected mutants could produce failures and program data states that are similar to those produced by real defects. Similar to that, Andrews et al. (2005, 2006) concluded in their studies that the detection ratio of mutants are representative for defect detection ratios, i.e., there is a correlation between them. One of the most recent papers that highlight the relationship between mutants and real defects was done by Just et al. (2014). They found a strong correlation between mutant detection ratios and real defect detection ratios.

Nevertheless, there are some works that identified limitations to the conclusions presented above. Namin and Kakarla (2011) found that there is only a weak correlation between mutants and defect detection ratios. Chekam et al. (2017) concluded that there is a strong connection between an increase of the mutation score and defect detection, but only at higher mutation score levels. A very recent paper by Papadakis et al. (2018) highlight, that there is only a weak correlation between the mutation score and defect detection, if the test suite size is controlled for.

As we can see on the above papers, the question if mutation testing is an appropriate tool to assess the defect detection capabilities of tests can not be definitely answered, as there exist indications for a positive as well as a negative answer to this question in the literature. While we also make use of mutation testing in our work, we do not create test suites, but reuse the ones that are provided by the projects. Hence, some limitations, e.g. that the test suite size must be controlled for, are not applicable to our research.

### *2.4. Defect classification*

There are numerous studies on defect classification. One of the main differences between these studies is the data on which the classification is based. While some taxonomies need specification or design documents (Hayes, 2003), others only need source code (Zhao et al., 2017), or defect reports (Xia et al., 2014; Tan et al., 2014).

A commonly used cause-driven taxonomy was proposed by Chillarege et al. (1992) and is called Orthogonal Defect Classification (ODC). In ODC defects are classified into eight types based on their description about the symptoms, semantics, and root causes. Offutt and Hayes (1996) proposed a model for the characterization of defects based on the syntactic and semantic size. Hayes (2003) presented a requirements-based defect analysis methodology, which the author also applied to NASA projects. Later, Hayes et al. (2005) developed two taxonomies, where the first classifies code modules (e.g., into data-centric, controller, view, ...) and the second code defects (e.g., into data, interface, computation, ...). Xia et al. (2014) used the descriptions available in defect reports to classify defects into two defect trigger categories (Bohrbug and Mandelbug) via natural language processing techniques. Tan et al. (2014) created a taxonomy in which defects are classified based on three dimensions: root cause (of the defect), impact (failure caused by the defect), and component (location of the defect).

The main problem with the approaches described above is that the data that is needed to classify the defects are often not available, e.g. specifications or design documents. This is especially true for open source projects, as these projects follow a special development process (Tu et al., 2000). Additionally, the creation of a link between the classified defect and its representation in the source code is often hard to achieve.

Recently, Zhao et al. (2017) presented an approach that can overcome the problems described above. They classify defects based on the change that was made to fix the defect. For this, they adapted the classification of Hayes et al. (2005) and created a tool for the C language that needs two versions of the source code of the program (defective and clean). Afterwards, the differences (changes) between these two versions are extracted and change patterns are detected. Based on these change patterns a change type classification is created, which is also the classification of the defect. They created five different categories with overall nine subcategories. The **data** category comprises of Changes on Data Declaration/Definition (CDDI) statements. This type of change indicate that a data-related defect occurred, e.g., if the declared type of a variable is changed from *int* to *float*. The **computation** category includes Changes on Assignment Statements (CAS). This includes the addition or deletion of assignment statements, or the modification of equations. Computation-related defects are defects that can lead to a wrong assignment of a variable, which are then often fixed by CAS. **Interface**-related defects are caused “by wrong definition or faulty function dependency on other functions." (Zhao et al., 2017). For example, if a function is called with an incorrect amount of parameters. Defects that are related to interface issues are fixed by Changes on Function Declaration/Definition (CFDD) or Changes on Function Call (CFC). The **logic/control** category comprises of Changes on Loop Statements (CLS), Changes on Branch Statements (CBS), and Changes on Return/Goto Statements (CRGS). Hence, defects that occur in these statements, e.g., changing a $<$ to a $>=$ in an if statement, "may cause the incorrect execution sequence or an abnormal state" (Zhao et al., 2017). Therefore, a defect-fixing change in these statements signals that a logic/control-related defect occurred. Every other change that could not be classified into the categories above is then subsumed under the **Other** category. For example, Changes on Preprocessor Directives (CPD), which is a change that can occur in programs using the C language.

Our approach reuses the classification scheme of Zhao et al. (2017) as it is tightly connected to the source code. This connection to the source code is needed to classify our generated mutants, as these mutants are not regular defects that exhibit, e.g., an issue report. While we reuse the logic behind every sub category our tool only outputs the main category in which a classified defect resides in (e.g., "Interface"). Furthermore, in contrast to Zhao et al. (2017), our tool is working for Java projects.

## 3. Research methodology

Within our RQ, we want to assess if unit or integration tests are more capable in detecting certain program defect types. Our expectation would be that integration tests detect more interface problems, while unit tests are more focused on finding computation defects or problems with the control flow of a program. This is also stated in software testing textbooks (Spillner et al., 2014; Ammann and Offutt, 2016; Myers et al., 2011). Hence, we evaluate if there are indeed defect types that are only (or mostly) detected by a certain test type and assess how capable these test types are in detecting certain defect types. These results could help to assess if the current definitions of the IEEE still fit to modern software development contexts.

Overall, our approach is as follows. We classify all test cases into unit and integration tests. Additionally, we collect data on the defect detection capabilities of each test case by using mutation analysis. We generate many different defective versions of the code and check which tests are able to detect the integrated defects. Afterwards, we classify our introduced defects based on the changes that were made to fix the defect using the taxonomy from Zhao et al. (2017). Finally, we evaluate which defect types are found by which test type. Within this section, we describe our case study design, including the subject selection, data collection, and analysis procedures.

### *3.1. Subject selection*

We defined the following inclusion criteria to select our study subjects.

- **Projects must be a library or a framework.** Libraries and frameworks should be included or used by other programs and are not executed by themselves. Therefore, tests that execute the whole system (system tests) are less likely to occur. System tests are used to detect problems in the design and architecture of the system by assessing the complete system (e.g., by giving an input to the *main* method and assessing that the computational steps taken are correct) (Ammann and Offutt, 2016). Libraries are less likely to have system tests, as they provide functionality through different entry points. For example, *commons-math* provides different functionality through different classes (e.g., you can do linear algebra or a numerical analysis, but both use different entry points into the library). Frameworks, on the other hand, can have a single entry point (e.g., parsers). However, they process an input and afterwards the user can call several functions on it. Hence, it is more likely that a more complex integration test is written for such a framework, which tests the parsing of the input plus the desired functionality. Hence, by focusing on libraries and frameworks we are reducing the risk of misclassifying a system test as an integration test, as our tooling infrastructure is currently not capable to differentiate between these test types.
- **Projects must have a minimum of 1000 commits and are at least 2 years old.** This ensures, that we only include mature projects.
- **Projects must use Java as programming language, Maven** (Apache Software Foundation, 2017b) **as build system, and** (JUnit Team, 2017) **or TestNG** (Beust, 2015) **as test driver.** This is a limitation of our current tooling infrastructure.

Moreover, one exclusion criterion is defined.

- **Projects should not be focused on Android alone.** This work concentrates on pure Java projects. While Android projects also use Java as programming language, there are several differences in testing those projects in contrast to pure Java projects.

**Table 1**
Selected projects with their characteristics.

| Project | Release | #Commits | #Java Classes | #Test Cases |
|---|---|---|---|---|
| commons-beanutils | 1.9.3 | 1128 | 370 | 1197 |
| commons-codec | 1.11 | 1677 | 166 | 874 |
| commons-collections | 4.1 | 2852 | 903 | 6637 |
| commons-io | 2.5 | 1868 | 308 | 1150 |
| commons-lang | 3.7 | 5106 | 711 | 4064 |
| commons-math | 3.6 | 5819 | 2596 | 6488 |
| druid | 1.1.0 | 5178 | 3785 | 4132 |
| fastjson | 1.2.41 | 2579 | 5214 | 4176 |
| google-gson | 2.8.0 | 1306 | 719 | 1014 |
| guice | 4.1 | 1513 | 2078 | 701 |
| HikariCP | 2.7.0 | 2493 | 154 | 120 |
| jackson-core | 2.9.0 | 1323 | 268 | 774 |
| jfreechart | 1.5.0 | 3622 | 1041 | 2175 |
| joda-time | 2.9 | 1913 | 527 | 4176 |
| jsoup | 1.11.1 | 1106 | 304 | 593 |
| mybatis-3 | 3.4.0 | 1816 | 1205 | 1053 |
| zxing | 3.3.0 | 3282 | 408 | 401 |

We applied these criteria on two different data sources. First, the list of Borges et al. (2017), which classified the most popular 5000 GitHub repositories (language-independent) into six different categories (application software, system software, web libraries and frameworks, non-web libraries and frameworks, software tools, and documentation). Second, we used a list of the most popular projects created by the maven repository (MVN Repository, 2018)[1] Overall, 41 projects fit to our criteria from which we randomly selected 17 projects for our study.

Table 1 gives an overview of the chosen projects together with their characteristics. It shows the project name, the release of the project that was used in the case study, the overall number of commits (till the stated release), the number of Java classes for the release, and the number of test cases that are executed by the build script of the release. All projects can be found on GitHub. The number of classes and test cases in this table highlight that the projects that we included in our case study are non-trivial projects.

From the sampled projects we included the latest minor release, if available. If this release was not available, we took the next possible release (e.g., release 1.11.1 of *jsoup*, as the release 1.11.0 is not

[1] As available on the time we performed our case study (January 2018).

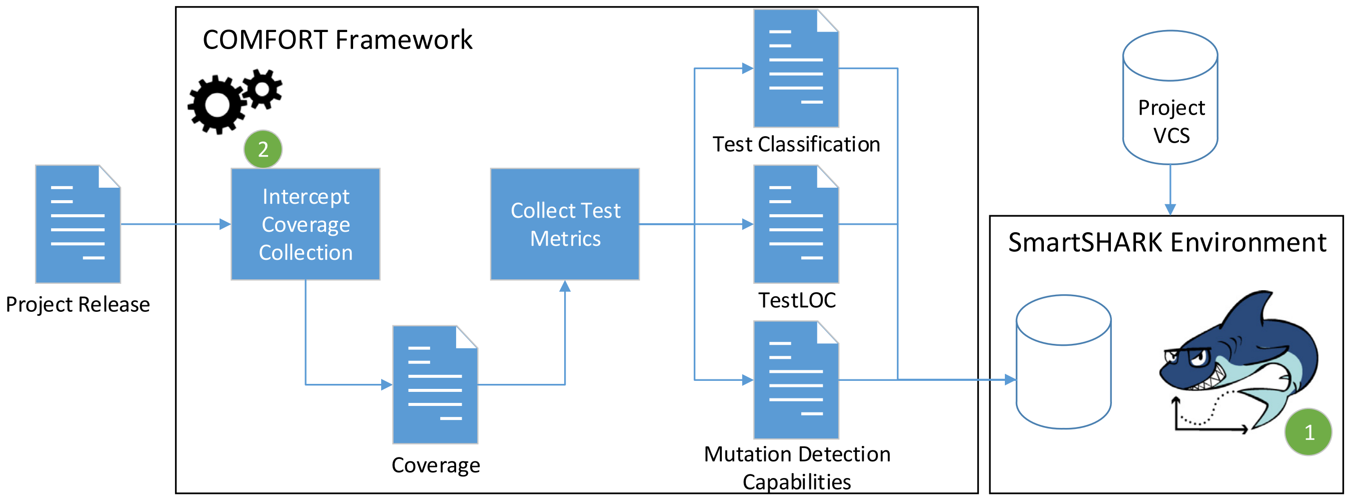

**Fig. 1.** Logical overview of our data collection process. Step 1 is the collection of meta- data using the SmartSHARK environment (Trautsch et al., 2016); Step 2 is the automatic collection of test metrics, including the mutation detection capabilities of all tests.

available). For two projects (i.e., *commons-beanutils* and *fastjson*) the latest minor release was longer than four years ago. Hence, we decided to use the most recent release to prevent compatibility issues with our infrastructure.

### 3.2. Data collection procedures

Fig. 1 gives a logical overview of our data collection process for our case study. As a first step, we are collecting meta-data about each project using the SmartSHARK environment (Trautsch et al., 2016). Hence, we collect data from the version control systems of the projects, as these are later on used in our data collection process to integrate the different collected data. Afterwards, in the second step, we calculate different test metrics for each analyzed test case. This step is automated by our Collection of Metrics for Tests (COMFORT) framework. In a first step, we execute the tests of a project release and intercept the coverage collection to collect the coverage on a *per test case level*. The resulting coverage is afterwards used to calculate different test metrics for each test case: the test classification into unit and integration test (see Section 3.2.1), the Test Lines of Code (TestLOC) (see Section 3.2.2), and the mutation detection capabilities (see Section 3.2.3). In the following, we describe each approach that we followed to calculate these metrics and the concrete implementation that we designed. Furthermore, we describe our approach for the collection of the defect type in Section 3.2.4.

#### 3.2.1. Assigning the test classification

Within this section, we explain our approach to assign a test class (i.e., unit or integration test) to the tests of the projects. We base our our separation on the IEEE definitions presented in Section 2.1, because the ISTQB definition (i.e., a unit test tests only *one* unit) is a subset of it. For example, all ISTQB unit tests are also IEEE unit tests by definition. Furthermore, our prior work (Trautsch and Grabowski, 2017) shows that the ISTQB definitions, are too restrictive for modern software systems, where *logical* software units often consist of several classes.

According to the definitions of the IEEE (see Section 2.1), a *test* in Java is a class, which consists of several test methods. These test methods are *test cases*, which are e.g., annotated with the “@Test”-JUnit annotation (JUnit Team, 2017). A unit is a *class* in Java, as it is a logically separable part of a Java program, which is not further subdivided into other components. This is in line with the literature (Borle et al., 2017). A unit test is a test case that tests “... software units or groups of related units” (IEEE, 2010). In Java, related units are put into one package, as the official Java documentation states: “A package is a namespace that organizes a set of related classes and interfaces.” (Oracle, 2017). Hence, if we apply the IEEE definition to Java a unit test is a test that tests only units from within one package (i.e., related units). On the other hand, an integration test tests units from more than one package. Hence, we do not classify the *intent* with which a test is created, but the actual type of the test according to the IEEE definitions.

To foster the understanding of the IEEE definitions, we selected two real world examples from the *commons-io* project (Apache Software Foundation, 2018). Listing 1 shows an IEEE unit test and an IEEE integration test. The upper part of this listing presents a typical unit test, where the correct workings of a function is tested. Here, the *getPrefix* function of the *FilenameUtils* class is tested. More precisely, this test checks if null bytes are part of the string. As this tests only calls one unit (i.e., the *FilenameUtils* class) and the *FilenameUtils* class does not call other units, this test gets classified as a unit test. On the other hand, the bottom part of the listing shows an IEEE integration test. In this case, the test checks if the *ByteArrayOutputStream* class of *commons-io* works as intended, if it is used within the *copy* function of the *CopyUtils* class. As the *ByteArrayOutputStream* is from the *org.apache.commons.io.output* package and the *CopyUtils* class from the *org.apache.commons.io* package, this tests gets classified as integration test.

On the implementation side, we use the recorded per test case coverage data for the test class assignment. A unit counts as covered, if at least one method of it was executed by the test. Furthermore, we only record the coverage of units that are *within* the project, i.e., calls to other libraries or the Java standard Application Programming Interface (API) or mocking frameworks are filtered out. In addition, we filter out every test class that might be inside the coverage data, as we want to base our test class assignment only on covered production classes. Therefore, we check the file path for each covered class and if the path has the term “test”[2] in it, we filter it out from the data. This has another advantage: self-made mocks that might exist in the project are filtered

[2] In Maven projects (like we use in our study) all tests and test related data is in the “src/test” folder.

```
// IEEE Unit Test
[...]
package org.apache.commons.io;
[...]

@Test
public void testGetPrefix_with_nullbyte() {
  try {
    assertEquals("~user\\", FilenameUtils.getPrefix("~u\
       u0000ser\\a\\b\\c.txt"));
  } catch (IllegalArgumentException ignore) {
  }
}

/*************************************************/

// IEEE Integration Test
[...]
package org.apache.commons.io;
[...]
import org.apache.commons.io.output.ByteArrayOutputStream;
[...]

@Test
public void copy_byteArrayToOutputStream() throws Exception {
  final ByteArrayOutputStream baout = new ByteArrayOutputStream
      ();
  final OutputStream out = new YellOnFlushAndCloseOutputStream(
      baout, false, true);

  CopyUtils.copy(inData, out);

  assertEquals("Sizes differ", inData.length, baout.size());
  assertTrue("Content differs", Arrays.equals(inData, baout.
      toByteArray()));
}
```

**Listing 1.** Examples for an unit test (top) and an integration test (bottom) from the *commons-io* project (Apache Software Foundation, 2018).

out from the coverage data. Within this filtered coverage data, we check how many “related units” (i.e., different packages) are covered by a test case. If only one logical unit is covered, then the test case is classified as unit test. Otherwise, it is classified as an integration test.

### 3.2.2. Collecting the TestLOC

We need to acquire the number of TestLOC for our RQ to normalize the results. This number is calculated based on the coverage data that is also used for the test classification. We sum up the number of covered lines for each test case, while we differentiate between covered production lines and covered test lines. This differentiation is done based on the path in which the covered class resides. If the path starts with "src/test" we know that the covered class is a test class, because it resides in the test folder, and is a production class otherwise.

In contrast to just counting Lines of Code (LOC) or Logical Lines of Code (LLOC) this approach has the advantage that we consider all executed test code than just the lines of the test case itself. Otherwise our results could be biased. For example, tests that have longer set up methods (e.g., to bring several units into a testable state) but less LOC would show better results after the normalization than tests where the whole set up is integrated into the test case itself.

### *3.2.3. Collecting the mutation detection capabilities*

We collect the mutation data for every test case that was executed (i.e., is available in the coverage data) by using PIT (Coles et al., 2016) within our framework. We execute PIT for every test case separately, i.e., for each test case that was covered. Each test case is run against all generated mutations. Afterwards, the results of PIT are parsed. If a test case failed when it is executed alone, e.g., because this test case depends on the execution of other test cases, we excluded it and no mutation data is collected. As the collection of the mutation data is very computation intensive (i.e., our case study experiment consumed about 35 months of CPU time) we executed it on the High Performance Computing (HPC) cluster hosted by the datacenter of our university[3]

We followed the best practices on using mutation testing in controlled experiments by Papadakis et al. (2017). Hence, we also report on each decision regarding the mutation analysis, as advised by Papadakis et al. (2017) in the following:

- **Mutation Testing Tool:** We used the paper by Kintis et al. (2017) as a basis for our decision regarding the mutation testing tool. We decided to use PIT (Coles, 2017) for our analysis due to several reasons. PIT possess a good defect-revelation ability Kintis et al. (2017), which is an indicator of its suitability for our experiments. Only $\text{PIT}_{\text{RV}}$ (Coles et al., 2016; Laurent et al., 2017) was able to reveal more defects (115:122). Nevertheless, we decided against $\text{PIT}_{\text{RV}}$, because the number of generated *equivalent* mutants is substantially higher for $\text{PIT}_{\text{RV}}$ in contrast to PIT Kintis et al. (2017). In fact in the experiment by Kintis et al. (2017), $\text{PIT}_{\text{RV}}$ generated about 88.74% more equivalent mutants than PIT. As equivalent mutants are a substantial threat to the validity of our study, we decided to use PIT instead of $\text{PIT}_{\text{RV}}$. We even contributed to the development of PIT by creating a pull request that got accepted and integrated into the version 1.3.2 of PIT. This addition was required for our work and allows PIT users to filter the test cases of tests against which the mutations are challenged.
- **Mutant Redundancy:** Mutant redundancy might have a large impact on the validity of our study. Hence, it is important to care for this kind of threat. We generated a lot of mutants for our case study ( $>$ 500.000). Checking them all by hand to detect duplicate mutants is a task that would take a substantial amount of time and is error prone. To mitigate this threat, we create the set of disjoint mutants, which only includes mutants that are not killed collaterally by most of the test cases (Kintis et al., 2017; Papadakis et al., 2016).
- **Mutant Selection:** We decided to use all mutation operators that are provided by PIT Coles (2017), including operators that are also used in integration mutation testing approaches (Delamaro et al., 2001; Grechanik and Devanla, 2016). The reason for this is that we wanted to generate a huge set of mutants so that we reduce the possibility that we only generate mutants of low quality or mutants that favor only one specific type of test. There is also empirical evidence that supports this approach (Kintis et al., 2017).
- **Test Suite Choice and Test Suite Size:** The test suite choice and its size is set, as we reuse the test suites that were created by the developers of the projects.
- **Clean Program Assumption:** The general problem that the Clean Program Assumption (CPA) describes is that test suites are assessed on the mutated program instead the original one for which they were created (Chekam et al., 2017). While we know of the CPA, we do not need to take it into account for our experiment, as we do not compare different testing techniques or rely on the coverage measured on the clean program.
- **Multiple Experimental Repetitions:** The calculations for our case study are only done once, as we do not have a component in it that make stochastic choices. The mutants that are generated for each test case and the result (i.e., mutant was killed or not) are not affected by any stochastic process. An exception is the creation of the disjoint mutant set, as the used algorithm is not deterministic. Hence, we repeated the generation of the disjoint mutant set 10 times.
- **Presentation of the Results:** As described above, our results are presented on the test case level.

### *3.2.4. Collecting the defect classification*

To gather the defect classification for each integrated mutation, we created the BugFixClassifier (Trautsch, 2018). The BugFixClassifier is able to extract changes between two files using CHANGEDISTILLER by Fluri and Gall (2006) and classify the detected changes based on the approach by Zhao et al. (2017). CHANGEDISTILLER (Fluri and Gall, 2006) is a tool that extract changes between two source code files based on the comparison of the Abstract Syntax Trees (ASTs) of both files. For each change that CHANGEDISTILLER detects, it outputs the change type (as defined by Fluri and Gall (2006)), which entity was changed (e.g., an if-statement or a method), and the parent entity of the changed entity (e.g., the initialization part of a for loop). We decided to reuse CHANGEDISTILLER, as it provides us with fine-grained source code changes that can be mapped onto the defect types as defined by Zhao et al. (2017). Hence, we do not only store each generated mutation, but also the type of defect that was integrated.

There is no common agreement in the literature on how to classify integrated mutants into different defect type classes. While there are other approaches, e.g. by Ghosh and Mathur (2001) or Delamaro et al. (2001) to generate specific interface mutants, we decided to use PIT and classify the mutants afterwards via the classification by Zhao et al. (2017). We decided for this approach, because we constitute the source code locality to the integrated defect in this way. Other approaches, e.g. (Delamaro et al., 2001), state that the propagation of a, via a mutation operator changed, value is more important. They classify a mutant as an interface mutant, if a value is changed that is later on used within a function call. However, it is not clear if these mutants are really affect the interface.

Zhao et al. (2017) determines the class of a defect based on the change that was made to fix the corresponding defect. The only change that we made to the classification schema is that we excluded the category CPD, because preprocessor directives are not available in Java. We reuse the detailed change types of Fluri and Gall (2006) and provide a mapping between them and the defect classes defined by Zhao et al. (2017). This mapping is only provided for the upper categories of our classification scheme (i.e., data, computation, interface, logic/control, others), as this level of detail is sufficient for our purpose. Table A.1 in Appendix A shows the

[3] See: http://gwdg.de.

**Table 2**
Mutation operators of PIT. Based on the table by Kintis et al. (2017).

| Mutation operator | Description |
|---|---|
| **AP**: *Argument Propagation* | $\{(nonVoidMethodCall(\dots, par),\ par)\}$ |
| **BFR**: *Boolean False Return* | Replaces primitive and boxed boolean return values with `false`. |
| **BTR**: *Boolean True Return* | Replaces primitive and boxed boolean return values with `true`. |
| **CB**: *Conditionals Boundary* | $\{(op_1, op_2) \mid (op_1, op_2) \in \{(<, <=), (<=, <), (>, >=), (>=, >)\}\}$ |
| **CC**: *Constructor Calls* | {(`new` *AClass*(), `null`)} |
| **EOR**: *Empty Object Return* | Replaces return values with an empty value for that type. For example, if the return type is a string it returns “”, if it is a `List` an empty `List` will be returned. |
| **I**: *Increments* | $\{(op_1, op_2) \mid op_1, op_2 \in \{++, --\} \wedge op_1 \neq op_2\}$ |
| **IC**: *Inline Constant* | $\{(c_1, c_2) \mid (c_1, c_2) \in$ {(1,0), ((`int`) x, `x+1`), (1.0, 0.0), (2.0, 0.0), ((`float`) x, 1.0), (`true`, `false`), (`false`, `true`)}} |
| **IN**: *Invert Negatives* | $\{(v, -v)\}$ |
| **M**: *Math* | $\{(op_1, op_2) \mid (op_1, op_2) \in \{(+,-), (-,+), (*, /), (/, *), (\%, *), (\&, \mid), (\wedge, \&), (<<, >>), (>>, <<), (>>>, <<)\}\}$ |
| **MV**: *Member Variable* | {(`member_var=...`, `member_var=b`) $\mid$ b $\in$ {`false`, 0, '\u0000', 0.0, `null`}} |
| **NR**: *Naked Receiver* | Replaces a method call with the receiver for non-void methods where the return type matches the receiver's type. |
| **NC**: *Negate Conditionals* | $\{(op_1, op_2) \mid (op_1, op_2) \in \{(==, !=), (!=, ==), (<=, >), (>=, <), (<, >=), (>, <=)\}\}$ |
| **NVMC**: *Non Void Method Calls* | {(nonVoidMethodCall(), c) $\mid$ c $\in$ {`false`, 0, 0.0, '\u0000', `null`}} |
| **NR**: *Null Return* | Replaces return values with null. |
| **PR**: *Primitive Return* | Replaces `int`, `short`, `long`, `char`, `float` and `double` return values with 0. |
| **RC**: *Remove Conditionals* | Removes or negates a conditional statement to force or prevent the execution of the guarded statements, e.g. {((a *op* b), true) *or* ((LHS && RHS), RHS)} |
| **RI**: *Remove Increments* | $\{(--v, v), (v--, v), (++v, v), (v++, v)\}$ |
| **RS**: *Remove Switch* | Changes all labels of the `switch` to the default one. |
| **RV**: *Return Values* | {(`return` a, `return` b) $\mid$ (a,b) $\in$ {(`true`, `false`), (`false`, `true`), (0, 1), ((`int`) x, 0), ((`long`) x, `x+1`), ((`float`) x, `-(x+1.0)`), (NAN, 0), (`non-null`, `null`), (`null`, `throw` RuntimeException)}} |
| **S**: *Switch* | Replaces the `switch`'s labels with the default one and vice versa (only for the first label that differs) |
| **VMC**: *Void Method Calls* | $\{(voidMethodCall(), \varnothing)\}$ |

mapping between the change types of Fluri and Gall (2006) and the defect classes of Zhao et al. (2017). Unfortunately, some change types of Fluri and Gall (2006) cannot be directly mapped onto a defect class. For these change types we need to consider the type of the changed entity and/or the type of the parent entity (e.g., METHOD, FOR_INIT). Table A.2 in Appendix A shows the conditions that need to be fulfilled for a change together with the defect class that is then assigned.

In general, the original non-defective and the defective source code is needed for the defect classification. The changes between both are then extracted and classified based on the rules above. Unfortunately, our used mutation testing framework does not offer the possibility to get the mutated source code, as the mutation is done on the byte-code of the original code (Coles et al., 2016). Nevertheless, some of the applied mutations can be directly mapped to a defect class. This mapping is presented in Table 3. For mutation operators that cannot be directly mapped (marked with “-” within Table 3) we used the following approach, using the BugFixClassifier. The mutation testing framework outputs the mutation operator that is used and the line in which it was used. Hence, we integrate the corresponding defect (based on the mutation operator) into the original source code in the specified line. Afterwards, we extract the changes between the original and the now defective source code to get the class of the defect that was integrated by the mutation operator. For example, assume that the math mutator (see Table 2) was used on line five of a Java file. Using this information, we can parse the original source file, look in line five to see which kind of operator is used originally on this line, and change it according to the rules of the math mutator. Afterwards, we can use this version of the code and the original one to create a defect class for the introduced defect.

**Table 3**
Mapping between the used mutation operators and the defect class.

| Mutation Operator | Defect Class |
|---|---|
| **AP**: *Argument Propagation* | Interface |
| **BFR**: *Boolean False Return* | Logic/Control |
| **BTR**: *Boolean True Return* | Logic/Control |
| **CB**: *Conditionals Boundary* | - |
| **CC**: *Constructor Calls* | Interface |
| **EOR**: *Empty Object Return* | Logic/Control |
| **I**: *Increments* | - |
| **IC**: *Inline Constant* | Data |
| **IN**: *Invert Negatives* | - |
| **M**: *Math* | - |
| **MV**: *Member Variable* | Computation |
| **NR**: *Naked Receiver* | Interface |
| **NC**: *Negate Conditionals* | - |
| **NVMC**: *Non Void Method Calls* | Interface |
| **NR**: *Null Return* | Logic/Control |
| **PR**: *Primitive Return* | Logic/Control |
| **RC**: *Remove Conditionals* | Logic/Control |
| **RI**: *Remove Increments* | - |
| **RS**: *Remove Switch* | Logic/Control |
| **RV**: *Return Values* | Logic/Control |
| **S**: *Switch* | Logic/Control |
| **VMC**: *Void Method Calls* | Interface |

## 3.3. Data analysis and results

In the following sections, we explain our analyzed data sets, our analysis procedures, as well as our results for our RQ.

### 3.3.1. Data sets

Table 4 shows the number of unit and integration tests for each project release that we analyzed, together with the number of not classifiable tests, the sum of the TestLOC for unit and integration tests, the number of unique mutants that are generated, and the number of analyzed tests. Overall, this table highlights that nearly all projects have more integration than unit tests. Nevertheless, for some projects our approach was not able to classify all test cases. We looked into all not classifiable test cases and found several reasons for this, e.g., tests that are empty, tests that directly return without executing any production code, or test cases that only test constants of classes. Furthermore, Table 4 highlights the number of generated unique mutants[4] together with the number of analyzed

[4] Not all mutants are generated for each test case as our mutation testing tool pre-selects mutations against which the test case should run by using the coverage data of the test case (Coles et al., 2016).

**Table 4**
Projects with their collected data, including Unit Tests (UT), Integration Tests (IT), and not classifiable tests (NC).

| Project | #UT | #IT | #NC | TestLOC (UT) | TestLOC (IT) | #Unique Mutants | #Analyzed Tests |
|---|---|---|---|---|---|---|---|
| commons-beanutils | 285 | 890 | 22 | 10,966 | 44,510 | 11,310 | 1175 |
| commons-codec | 707 | 146 | 21 | 6160 | 1136 | 9059 | 853 |
| commons-collections | 1925 | 4005 | 707 | 63,048 | 260,958 | 26,464 | 5930 |
| commons-io | 634 | 504 | 12 | 17,345 | 13,185 | 9593 | 1138 |
| commons-lang | 3507 | 471 | 86 | 36,718 | 10,841 | 36,549 | 3978 |
| commons-math | 775 | 5709 | 4 | 10,812 | 157,902 | 113,469 | 6484 |
| druid | 94 | 4037 | 1 | 585 | 71,456 | 123,835 | 4127 |
| fastjson | 315 | 3842 | 19 | 1968 | 41,312 | 48,289 | 4147 |
| google-gson | 215 | 797 | 2 | 2185 | 11,115 | 8816 | 1012 |
| guice | 28 | 673 | 0 | 265 | 13,526 | 10,851 | 688 |
| HikariCP | 21 | 97 | 2 | 291 | 5848 | 4967 | 117 |
| jackson-core | 47 | 727 | 0 | 490 | 17,567 | 34,361 | 774 |
| jfreechart | 228 | 1946 | 1 | 2642 | 31,240 | 96,104 | 2174 |
| joda-time | 179 | 3975 | 22 | 2810 | 83,739 | 32,951 | 4153 |
| jsoup | 64 | 525 | 4 | 705 | 11,772 | 14,171 | 588 |
| mybatis-3 | 224 | 824 | 5 | 1449 | 28,577 | 17,126 | 1043 |
| zxing | 108 | 293 | 0 | 1564 | 13,809 | 29,592 | 401 |
| Overall | 9356 | 29,461 | 908 | 160,003 | 818,493 | 627,507 | 38,782 |

tests. The number of analyzed tests can be lower than the number of all test cases of the projects, as our mutation testing tool might not be able to run the test alone (e.g., test cases that fail if they are executed alone).

We create two different data sets for the analysis of our research question based on the collected data. These data sets represent different perspectives on the questions at hand.

- **ALL:** This data set consists of the test results for all generated mutations. With this data set we want to assess the defect detection capabilities of unit and integration tests for a large data set with many different defects that are integrated.
- **DISJ:** This data set consists of the test results for the set of disjoint mutants (see Section 3.2.3). With the analysis of this data set we can gain insights into the defect detection capabilities of unit and integration tests for defects that are hard to kill (Papadakis et al., 2016).

### 3.3.2. Analysis procedure

For all of the in Section 3.3.1 presented data sets we executed the following analysis process.

1. **Calculate the number of detected defects:** We gather the number of all detected defects by summing up the number of defects that are detected by a unit or an integration test. We consider a mutation that is *killed* as a detected defect. The results for test cases that are executed with several different parameters, are combined and analyzed as one test case. As the algorithm applied to create the set of disjoint mutants is not deterministic, we repeat the analysis process using the disjoint mutant set 10 times and take the average of all 10 runs for the number of detected defects.
2. **Build the sum for each test type:** We divide the detected defects by their type and sum them up for unit and integration tests separately. This way, we can assess if unit or integration tests are more capable in detecting a certain kind of defect. We excluded the defect type *Other* from our analysis, as it does not represent a real defect type, but more a type of change that can not be classified as one of the other types (see Section 2.4). Note, that we have prespecified the subgroup analysis for this paper. Hence, it is not a post hoc analysis, which would threat the validity of our study (Wang et al., 2007).
3. **Normalize by the number of TestLOC:** The resulting sums from the previous step are normalized by the number of TestKLOC to create scores. Hence, the result is the number of detected defects per 1000 TestLOC. We perform this normalization step because we want to include the effort that is done to create tests into our analysis. Hence, this normalization step is needed, as we would otherwise just compare the number of detected defects, which would ignore the effort (i.e., the TestLOC).
4. **Statistical Testing:** As a last step, we take the normalized values of each project and perform statistical tests to check if the differences between the unit and integration test scores are significant. Hence, we do a statistical test for each defect type using the calculated scores of unit and integration tests of each project. We first check if our populations follows a normal distribution (applying the Shapiro-Wilk test Shapiro and Wilk (1965)) and have equal variances (applying the Levene test Olkin (1960)) to choose an appropriate significance test. Based on the results, we either perform a students *t*-test (Student, 1908) or a Mann-Whitney-*U* test (Mann and Whitney, 1947). We use a significance level of $\alpha = 0.05$ for all of our statistical tests. As we are performing multiple statistical tests, which could increase the overall change of false discoveries (Type 1 Errors) (Aickin and Gensler, 1996), we need to apply corrections for multiple comparisons. We decided to use the Bonferroni correction (Aickin and Gensler, 1996) for all of our statistical hypothesis tests that we made. Overall, we use 8 statistical hypothesis tests: we use our **ALL** and **DISJ** data sets and check for differences in the scores for each defect type (i.e., COMPUTATION, DATA, INTERFACE, LOGIC/CONTROL), which results in eight different tests (two data sets * 4 defect types). Therefore our adjusted significance level is $\alpha^* = 0.05/8 = 0.00625$.

### 3.3.3. Results

Tables 5, and 6 show the scores (i.e., number of detected defects per TestKLOC) of unit and integration tests, separated by the type of defect that they have detected. Additionally, the mean and standard deviation is shown for each column. Additional tables including the number of defects found by unit tests, integration tests, and both for each defect type are available in the supplementary material (see Section 3.4). Furthermore, the supplementary material includes Venn-Diagrams for each project to visualize these numbers.

Table 5 depicts the results for the data set **ALL**. It shows that if we separate the integrated defects by type, the mean scores of integration tests are higher than the mean scores of unit tests for

**Table 5**
Scores for unit and integration tests for the **ALL** data set, separated by defect type.

| | COMP. | | DATA | | INT. | | L/C | |
|---|---|---|---|---|---|---|---|---|
| Project | UT | IT | UT | IT | UT | IT | UT | IT |
| commons-beanutils | 3.83 | 2.29 | 6.29 | 3.30 | 38.39 | 32.44 | 69.31 | 44.69 |
| commons-codec | 71.27 | 22.01 | 134.42 | 66.02 | 252.44 | 344.19 | 389.12 | 448.94 |
| commons-collections | 2.24 | 1.25 | 2.98 | 1.11 | 8.31 | 7.05 | 20.08 | 13.62 |
| commons-io | 11.01 | 15.09 | 20.12 | 19.34 | 57.94 | 64.77 | 101.64 | 104.29 |
| commons-lang | 24.59 | 17.06 | 61.36 | 44.46 | 125.25 | 162.62 | 354.29 | 276.36 |
| commons-math | 28.39 | 38.68 | 41.90 | 76.47 | 86.57 | 129.81 | 195.15 | 215.08 |
| druid | 133.33 | 57.56 | 169.23 | 55.92 | 194.87 | 396.17 | 558.97 | 501.95 |
| fastjson | 31.50 | 50.47 | 66.57 | 109.65 | 63.01 | 213.72 | 263.72 | 328.86 |
| google-gson | 66.36 | 17.09 | 59.04 | 14.22 | 65.45 | 115.61 | 174.37 | 157.17 |
| guice | 11.32 | 37.93 | 18.87 | 31.05 | 226.42 | 223.94 | 362.26 | 264.82 |
| HikariCP | 37.80 | 30.61 | 65.29 | 28.73 | 151.20 | 125.68 | 109.97 | 163.13 |
| jackson-core | 87.76 | 153.07 | 136.73 | 154.32 | 293.88 | 195.42 | 561.22 | 589.97 |
| jfreechart | 36.34 | 76.47 | 71.16 | 92.93 | 75.32 | 290.01 | 281.98 | 593.41 |
| joda-time | 7.47 | 13.76 | 19.93 | 21.91 | 65.12 | 100.98 | 87.19 | 158.89 |
| jsoup | 5.67 | 32.45 | 19.86 | 58.78 | 41.13 | 280.33 | 114.89 | 372.07 |
| mybatis-3 | 15.87 | 13.89 | 18.63 | 12.25 | 193.24 | 119.15 | 204.28 | 116.21 |
| zxing | 41.56 | 81.83 | 190.54 | 211.46 | 203.32 | 282.42 | 435.42 | 611.12 |
| Mean | 36.25 | 38.91 | 64.88 | 58.94 | 125.99 | 181.43 | 251.99 | 291.80 |
| StDev | 35.45 | 37.67 | 58.56 | 56.96 | 86.22 | 110.24 | 168.34 | 197.19 |

**Table 6**
Scores for unit and integration tests for the **DISJ** data set, separated by defect type.

| | COMP. | | DATA | | INT. | | L/C | |
|---|---|---|---|---|---|---|---|---|
| Project | UT | IT | UT | IT | UT | IT | UT | IT |
| commons-beanutils | 0.09 | 0.00 | 0.00 | 0.02 | 0.18 | 0.09 | 0.64 | 0.38 |
| commons-codec | 1.30 | 0.88 | 0.65 | 0.00 | 2.44 | 2.64 | 7.31 | 2.64 |
| commons-collections | 0.02 | 0.00 | 0.02 | 0.01 | 0.13 | 0.04 | 0.40 | 0.11 |
| commons-io | 0.35 | 0.15 | 0.23 | 0.08 | 1.56 | 0.15 | 1.50 | 0.76 |
| commons-lang | 0.44 | 0.00 | 0.93 | 0.18 | 2.31 | 0.46 | 8.14 | 0.46 |
| commons-math | 0.28 | 0.28 | 0.00 | 0.16 | 0.37 | 0.16 | 1.85 | 1.04 |
| druid | 0.00 | 0.64 | 6.84 | 0.34 | 6.84 | 1.75 | 17.09 | 2.70 |
| fastjson | 0.51 | 1.02 | 0.00 | 1.19 | 0.00 | 2.52 | 1.52 | 6.00 |
| google-gson | 0.92 | 0.09 | 0.00 | 0.27 | 0.00 | 0.09 | 0.92 | 0.81 |
| guice | 3.77 | 0.30 | 0.00 | 0.44 | 0.00 | 1.11 | 0.00 | 1.85 |
| HikariCP | 0.00 | 0.00 | 0.00 | 0.17 | 0.00 | 0.17 | 0.00 | 0.17 |
| jackson-core | 0.00 | 0.23 | 2.04 | 0.06 | 0.00 | 0.17 | 10.20 | 0.91 |
| jfreechart | 1.89 | 0.29 | 0.38 | 0.10 | 0.76 | 0.54 | 5.30 | 1.28 |
| joda-time | 0.00 | 0.12 | 0.00 | 0.16 | 0.71 | 0.57 | 1.42 | 2.67 |
| jsoup | 0.00 | 0.00 | 0.00 | 0.08 | 0.00 | 0.08 | 0.00 | 0.08 |
| mybatis-3 | 0.69 | 0.14 | 0.69 | 0.00 | 1.38 | 0.07 | 0.00 | 0.03 |
| zxing | 0.64 | 0.00 | 0.00 | 0.14 | 0.00 | 0.07 | 0.64 | 0.29 |
| Mean | 0.64 | 0.24 | 0.69 | 0.20 | 0.98 | 0.63 | 3.35 | 1.31 |
| StDev | 0.97 | 0.31 | 1.67 | 0.28 | 1.72 | 0.86 | 4.78 | 1.52 |

each defect type except DATA defects, while they have a higher standard deviation. Hence, on average it seems that integration tests are more capable (i.e., scores are higher) in detecting defects for any defect type, except DATA defects. Furthermore, Table 5 also highlights the differences between projects. For some projects the results are as expected (i.e., integration tests are more capable in detecting interface defects, while unit tests are more capable in detecting other defects), while for other projects it is vice versa (e.g., *jackson-core*).

Table 6 highlights the results for the **DISJ** data set. It shows a different picture than Table 5. For the disjoint mutant set our results show that (on average) integration tests are less capable in detecting any type of defect than unit tests, i.e., the mean of integration test scores are lower than the mean of unit test scores for any of the defect types. But, the standard deviation for integration test scores are lower than the standard deviation of unit test scores for any of the defect types. Nevertheless, Table 6 also highlights, that some mutations are only detected by integration tests (e.g., interface defects for the projects *fastjson, google-gson, guice, HikariCP, jackson-core*) or only unit tests (e.g., computation defects for the projects*commons-beanutils, commons-lang*).

**Table 7**
P-values of the significance tests that were performed between the scores of unit and integration tests for the data sets **ALL** and **DISJ** and each defect type.

| Defect Type | ALL | DISJ |
|---|---|---|
| COMPUTATION | $p = 0.352$ | $p = 0.152$ |
| DATA | $p = 0.766$ | $p = 0.220$ |
| INTERFACE | $p = 0.112$ | $p = 0.278$ |
| LOGIC/CONTROL | $p = 0.531$ | $p = 0.267$ |

Table 7 presents the p-values of the significance tests that were performed between the scores of unit and integration tests for the data sets **ALL** and **DISJ** and each defect type. It highlights, that there are no statistical significant differences in the defect detection capabilities of unit and integration tests for any defect type.

**Answer to our RQ:** Tables 5 and 6 highlight, that there are differences in the defect detection capabilities between unit and integration tests for the different defect types if we compare the means in the tables. The results which test type is more capable in detecting which defect type is not consistent over our data sets, indicating that unit tests are more capable in detecting “hard to kill” defects. Furthermore, the differences in the defect detection capabilities are not significant, as Table 7 highlights. Our results also vary from project to project: there are some projects which seem to have more capable unit tests for certain defect types, while in others integration tests are more capable for the same defect type.

### 3.4. Replication kit

To facilitate further insights and the replication of our study, we provide a replication kit (Trautsch et al., 2018). The replication kit contains the following data:

- all source code used for the data collection, including the used versions of the COMFORT framework, the BugFixClassifier, and the used tools of the SmartSHARK environment;
- all source code used for the analysis of the collected data;
- a MongoDB database dump with all raw results, except the developer information; and
- additional visualizations of the results.

## 4. Discussion

The results of our research were rather unexpected for us, as it does not represents what we have learned and what we teach at our university. Overall, we could not find any significant differences in the defect detection capabilities between unit and integration tests for any of the data sets that we tested. These results are interesting out of several perspectives.

**Education:** Academia, as well as organizations like the ISTQB, teach that integration and unit tests are equally important and find different types of defects (i.e., integration tests detect interface defects, whereas unit tests detect other kind of defects). They also highlight that a separation between unit and integration tests make sense and should be done. Nevertheless, our results show that there is no significant difference in the defect detection capabilities between unit and integration tests for any of the defect types. Hence, it seems that unit and integration tests are even equally capable in detecting interface defects. In addition, we have seen that there are defects that are detected by both test types. These results contradict the accepted belief that unit and integration tests detect different types of defects in the software. This raises the question, if those definitions still fit to modern development contexts or if we should apply another distinction criterion to differentiate between unit and integration tests. It might not be a good idea to distinguish tests based on their properties, but on their usage as (Ammann and Offutt, 2016) suggest. They reason that “most of the literature emphasizes these levels in terms of **when** they are applied, a more important distinction is on the **types of faults** that we are looking for.” (Ammann and Offutt, 2016). Hence, (Ammann and Offutt, 2016) suggest that tests should be categorized according to their purpose or usage (i.e., types of defects targeted) and not according to their properties or when they are applied. While they use a different terminology, the core of the separation between unit and integration tests is similar to the definitions of the IEEE. This paper provides empirical data that it might not make sense to differentiate unit and integration tests in the way we nowadays do. Instead, we should create new definitions that fit to the modern software development contexts and should follow the proposal of Ammann and Offutt (2016) to separate tests based on which types of defects are targeted. Hence, we would need to revise current valid definitions that we teach in academia and industrial certifications. Our study results can provide valuable insights that could help with this revision.

**Practice:** Software development should always be accompanied by testing the developed parts. Our results show, that there seems to be no need to focus on the type of test developed, at least if we only consider the defect detection capabilities of tests and classify them according to the IEEE definition. Nevertheless, other influences could play a role, e.g., if a test should serve as documentation (Demeyer et al., 2002) or the maintainability of a test (Athanasiou et al., 2014). Our results highlight, that there are differences from project to project. Hence, it seems that the context in which a project is developed has an influence on the defect detection capabilities of the test types, as for some projects unit tests are more capable than integration tests, while for others it is vice versa. However, it seems that it is more important that developers “just create tests” instead of caring about mocking classes or which test type is developed.

To come back to the cited discussion from our introduction, where developers discuss if they should change their testing habit and focus more on integration than unit testing: our results show that this change would not have a negative impact, as both test types are equally capable. Hence, if developers argue that integration tests are more realistic and valuable in their daily developer life, our results do not argue against this practice. However, we also found that some defects were only detected by either unit or integration tests. Moreover, our results highlight that unit tests are better capable (on average) to kill mutants that are hard to kill, as the results with the **DISJ** data set depict. Hence, our results show that it is still favorable to test on both test levels.

## 5. Towards new definitions

After analyzing the data from our first paper Trautsch and Grabowski (2017), this paper, and discussions that we had with software engineers, we concluded that we need new definitions for unit and integration testing. However, to come up with new definitions that work will all kind of systems (e.g., modern web-based systems and legacy systems) is a hard task that needs a lot of time and discussions with a broader audience from research and industry. Nevertheless, we want to provide a starting point, based on our results, our experience, and our discussions that we had, from which we can start to debate.

There are several requirements that the new definitions must fulfill: first, they should be easy understandable and applicable (also) to modern software development environments, as this is one of the shortcomings of the current definitions, as our results highlight; second, they should work with a broad range of different kinds of systems, so that they are not too specific to one use-case; third, they should form a clear separation between unit and integration tests, as developers are often confused about what a unit or an integration test is Trautsch and Grabowski (2017).

In our opinion, the current definitions for unit and integration tests are too fine-granular for the modern software development. This is mostly due to the fine-grained definition of the term unit by current standards. Hence, our solution would be to make this definition more coarse-granular: we would define a unit as a component of a complex software system, e.g., a webservice or the frontend of the system (see Fig. 2). Hence, a unit test would be a test that only assesses parts that are within one component, e.g., a test that checks the interface of a webservice (i.e., if it react correctly or returns the correct data for a given input). But, if a test assesses, e.g., the frontend and the frontend gets data from a webservice, we would define it as an integration test, as this test now

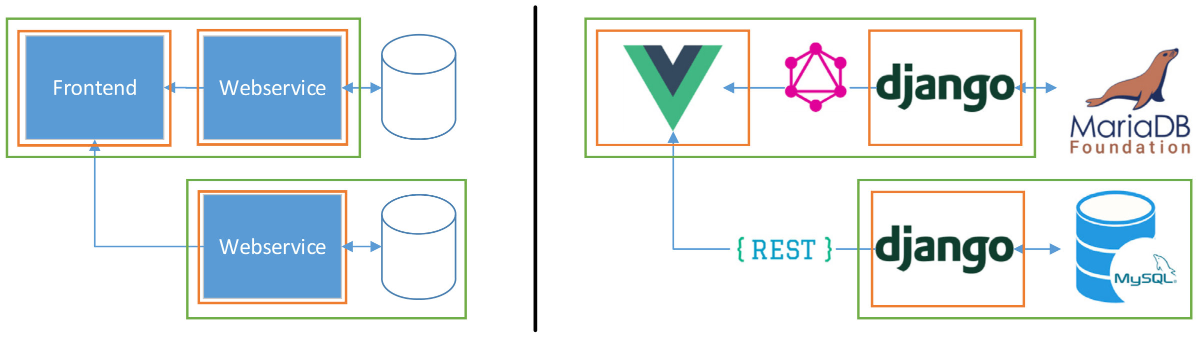

**Fig. 2.** Left side: Abstract example of a software system, consisting of two different web services, two databases and one frontend. Right side: Concrete instantiation of the example of the left side. The frontend (realized via Vue.js You (2019)) is communicating with two different web services implemented via Django Django Software Foundation (2019) over GraphQL Facebook Open Source (2019) and REST Fielding and Taylor (2000). The orange boxes show potential unit tests, while the green boxes highlight potential integration tests. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

needs two different software components to work together. Furthermore, we are testing if these interfaces between those units work well with each other, or if, e.g., the frontend calls the webservice wrong or expects different data.

This idea is also depicted in Fig. 2. On the left side, we see an abstract example of a software system, consisting of three different units (two webservices and a frontend that uses both of them) and two databases. The orange boxes show potential unit tests: if a test only tests part from the frontend, without the need to call the webservices, it is called a unit test. Otherwise, the green boxes highlight potential integration tests: if we want to test if the frontend can process the data coming from the first webservice, and we use both of these units, we define it as an integration test. Furthermore, if we want to assess if a webservice is working well with a database and use functions from within the webservice and the database, we would also call this an integration test. On the right side of Fig. 2 there is an instantiation of the scenario depicted on the left side: here, we exchanged the abstract parts with concrete technologies that are currently used within complex software systems (e.g., REST interfaces, GraphQL interfaces, Javascript single page applications).

Our new definition has several advantages. As explained above, such a definition should be easily understandable and applicable. We believe that our definition of unit and integration tests are better understandable, as it is clear that a unit test only assesses parts from within one software unit, while an integration test assesses several parts. The old definition is similar, but more fine-granular, which makes it difficult for developers to decide what type of test she has written Trautsch and Grabowski (2017). Furthermore, we suspect that our definitions are closer to the software development reality: the current more fine-granular definitions do not fit into the current development context, as our results highlight. We could not make out any difference between unit and integration tests in respect to their defect detection capabilities. Our new definitions would be more suitable for a distinction in this case. However, our proposed definitions have some problems. There might be systems that may consist of only one component. Hence, by our definition it would only have unit tests. To mitigate this kind of problem, we propose another testing type, which we call ”algorithm test”. An algorithm test is a test that tests only one function or method of a software system. This idea stems from some comments of software engineers that said, that they mostly write integration tests, but for some important or complex algorithms, they write tests that are focused only on this specific algorithm. Hence, this would provide us another layer of abstraction, besides unit and integration tests. Another problem with our definitions is that we do not know if they really work with all kind of systems. We believe that they work well with modern software systems, but we do not have data for legacy systems. Furthermore, there are some other implications of our definitions that need more research. Our new definition of a unit test would allow us to state in which software unit the problem resides in, but not in which class/module. However, we need more research to evaluate if this is important in our modern software development world, or if it is enough that we can make out the software component in which the defect resides in, as a developer would run the test via a debugger anyway. Furthermore, another current distinction between unit and integration tests is the execution time that a test needs: unit tests should be very fast Saff and Ernst (2004), while integration tests might have dependencies that make the execution time slower (e.g., databases). With our new definitions, this distinction would be the same: unit tests would be faster than integration tests. However a unit test would be slow, if a developer decides that she want to write one single test that assesses the complete software unit. Moreover, the new definitions that we proposed are more ideas that are open for discussion, as they are not fully-fledged definitions. We propose one way to think about this to bring us into the direction to create new definitions that might be more fitting for modern software development.

## 6. Threats to validity

In this section, we discuss the external, internal, and construct validity of our study together with the validation procedures that we have taken to counter or measure those threats.

### *6.1. Construct validity*

Threats to this type of validity are concerned with the degree to which our analysis really measures what we intended it to measure. While we carefully tested our tools and scripts via manually written tests and manually curated samples of data, there can still be defects, which can influence our results.

For the collection of the mutation data we make use of PIT (Coles et al., 2016), which is a mature mutation testing framework that is often used in research, e.g. (Kintis et al., 2017; Laurent et al., 2017). Hence, it is less likely that the mutation testing is not working correctly. Every problem that occurred during the mutation testing was manually inspected. We found that PIT sometimes reported that a test was not finishing without a failure, but this only occurred for 35 out of 36,435 test cases. Furthermore, PIT is designed for mutation testing on unit level. While PIT offers some mutation operators that are also offered by tools that are used for integration mutation testing (e.g. Delamaro et al. (2001); Grechanik and Devanla (2016); Ghosh and Mathur (2001)), this still could have an influence on our results. Nevertheless, PIT is constructing mutants that affect the interface of a unit. These mutants can be found by unit, as well as integration tests. However, the strategy with which these mutants are generated might not be optimal. Other research (e.g., Delamaro et al. (2001)) developed different strategies and mutation operators that are directly designed for interface mutations, which are not implemented in PIT.

The threat of mutant redundancy is substantial for our kind of analysis. We tried to reduce this thread by including the disjoint mutant set into our analysis process, as proposed by Kintis et al. (2017). Nevertheless, this is an approximation and it might not be the most fitting set of mutants.

Another threat is our choice of the defect classification scheme. A different classification scheme might produce different results for our research question. But, we have chosen the scheme by Zhao et al. (2017) as it is close to the code, which is needed to classify integrated mutations, and it provides a good overview of different defect types.

Moreover, our approach to create a defect classification for an integrated mutation can be flawed for some mutations. For example, if a mutation is integrated in a line which has several statements on it so that the re-integration of the defect in the correct source code could be done in the wrong statement. To measure this threat, we manually checked a sample of our defect type classifications. None of the defect type classifications had the problem mentioned above.

We reuse the build file of the projects that we analyze to execute all of their tests. Hence, it can be the case that some tests are filtered out, e.g., because they should only be executed via an continuous integration system as they are very long running. It can be the case that these filtered tests are always tests from the same type (e.g., integration tests, as they often have a higher execution time than unit tests). Nevertheless, we assess this threat by counting the number of tests that are excluded from execution. Overall, 67 tests are excluded. However, this number is rather low in comparison to the overall number of executed tests (i.e. 38782).

### 6.2. Internal validity

Internal validity threats are concerned with the ability to draw conclusions from the relation between causes and effects. While we tried to create an isolated environment where we have control over influencing variables (e.g., by using mutation analysis to integrate defects into a software) it can be the case that the defect detection capabilities are also influenced by other variables. We countered the influences that we know of, e.g., by normalizing the results.

The current version of our framework can not differentiate integration from system tests. Hence, it is possible that a system test is classified as an integration test. To mitigate this issue we included only projects that are libraries or frameworks.

The statistical tests used in this paper rely on the accurate implementations of the algorithms in the external library used. To ease this threat we are using a well known public library, namely SciPy (SciPy developers, 2017), for these algorithms.

### 6.3. External validity

Threats to this type of validity are concerned with the ability to generalize our results. We had a look at Java projects only. Hence, the results can be different for projects written in other languages. Although we analyzed a larger sample of projects than most other related work, the results can vary if other projects are chosen. Especially, as our set of projects is limited to open-source projects, even if some of the projects are developed by companies in an open-source manner (e.g., *google-gson*). Furthermore, we only selected libraries or frameworks which could potentially influence our results. Hence, our results could be different for other project types (e.g., applications). But, our approach needs a compilable release of a project, which is often not given as Tufano et al. (2017) highlight in their paper. This complicates a fully automatic analysis with more data. Nevertheless, the replication of this work using other projects (and other programming languages) is required in order to reach a more general conclusion. Hence, we added a replication kit that includes all data and programs of this paper to support such conceptual replications.

## 7. Conclusion and future work

In this paper, we reported an empirical study that was conducted on 17 open-source Java projects. The goal of this study was to investigate, if the existing standard definitions of unit and integration tests are still valid in modern software development contexts. We created two different data sets to pursue this goal: mutation testing data representing a large variety of defects that could be introduced in the program (627507 unique mutations) and the disjoint mutation testing data set that represents only “hard to kill” mutants. Additionally, we classified all created mutations into different defect types to evaluate if one test type is better capable in detecting a certain defect type. Overall we collected and analyzed the defect detection capabilities of 38,782 test cases.

Our results highlight, that there are no significant differences in the defect detection capabilities of unit and integration tests in either of our two data sets. However, we also found that there are some defects that can only be detected by one test type (i.e., either unit or integration tests). Furthermore, we found that we can not state that one unit tests or integration tests are more capable in detecting a certain defect type, as our tests found no statistical significant differences. Hence, it seems that the current standard definitions do not fit to modern software development contexts anymore, as there should be a difference between those test types. Therefore, our results questions if the division between unit and integration tests is reasonable for modern systems, like we develop nowadays. These results suggest that we should create usage-based definitions instead of property-based definitions of unit and integration tests so that they fit to modern software development contexts.

Our future work includes but is not limited to the use of more projects with different programming languages for the performed and additional analyses. As additional analysis we plan a qualitative study on the defects and test cases that we used in this study. This could help us to understand the differences between unit and integration tests and their (not) detected defects. In addition, we want to perform a deeper analysis for other aspects in which unit and integration tests might differ, e.g., the maintainability (Athanasiou et al., 2014) to get a more holistic view on the matter at hand. Furthermore, we would like to perform a developer study on unit and integration testing practices to get feedback from developers how they use unit and integration tests in their daily work. In connection to this study, it would be interesting to assess the usage of different test types in different development phases or development models, e.g., by comparing the usage of unit and integration tests in an classical and agile development environment.

**Acknowledgements**

The authors would like to thank the GWDG for the access to their HPC resources.

## Appendix A. Change Type Mapping

**Table A.1**
Mapping of the change types by Fluri and Gall (2006) that can be *directly* mapped onto the defect classes by Zhao et al. (2017).

| Change Type | Defect Class |
|---|---|
| ADDING_ATTRIBUTE_MODIFIABILITY | Data |
| ADDING_CLASS_DERIVABILITY | Interface |
| ADDING_METHOD_OVERRIDABILITY | Interface |
| ADDITIONAL_CLASS | Interface |
| ADDITIONAL_OBJECT_STATE | Data |
| ALTERNATIVE_PART_DELETE | Logic/Control |
| ALTERNATIVE_PART_INSERT | Logic/Control |
| ATTRIBUTE_RENAMING | Data |
| ATTRIBUTE_TYPE_CHANGE | Data |
| CLASS_RENAMING | Interface |
| COMMENT_DELETE | Other |
| COMMENT_INSERT | Other |
| COMMENT_MOVE | Other |
| COMMENT_UPDATE | Other |
| CONDITION_EXPRESSION_CHANGE | Logic/Control |
| DECREASING_ACCESSIBILITY_CHANGE | Interface |
| DOC_DELETE | Other |
| DOC_INSERT | Other |
| DOC_UPDATE | Other |
| INCREASING_ACCESSIBILITY_CHANGE | Interface |
| METHOD_RENAMING | Interface |
| PARAMETER_DELETE | Interface |
| PARAMETER_INSERT | Interface |
| PARAMETER_ORDERING_CHANGE | Interface |
| PARAMETER_RENAMING | Interface |
| PARAMETER_TYPE_CHANGE | Interface |
| PARENT_CLASS_CHANGE | Interface |
| PARENT_CLASS_DELETE | Interface |
| PARENT_CLASS_INSERT | Interface |
| PARENT_INTERFACE_CHANGE | Interface |
| PARENT_INTERFACE_DELETE | Interface |
| PARENT_INTERFACE_INSERT | Interface |
| REMOVED_CLASS | Interface |
| REMOVED_OBJECT_STATE | Data |
| REMOVING_ATTRIBUTE_MODIFIABILITY | Data |
| REMOVING_CLASS_DERIVABILITY | Interface |
| REMOVING_METHOD_OVERRIDABILITY | Interface |
| RETURN_TYPE_CHANGE | Interface |
| RETURN_TYPE_DELETE | Interface |
| RETURN_TYPE_INSERT | Interface |

**Table A.2**
Mapping of the change types (CT) by Fluri and Gall (2006), where the changed entity (CE) and/or the parent entity (PE) needs to be taken into account to map a change onto the defect classes by Zhao et al. (2017). The term STATEMENT_* includes the general change types, i.e., STATEMENT_UPDATE, STATEMENT_INSERT, STATEMENT_DELETE, STATEMENT_PARENT_CHANGE, STATEMENT_ORDERING_CHANGE.

| Condition | Defect Class |
|---|---|
| $CT \in \{\mathtt{STATEMENT\_*}\} \wedge$ $CE \in \{\mathtt{ASSIGNMENT,\ POSTFIX\_EXPRESSION,\ PREFIX\_EXPRESSION}\} \wedge$ $PE \notin \{\mathtt{FOR\_INCR}\}$ | Computation |
| $CT \in \{\mathtt{STATEMENT\_*}\} \wedge$ $CE \in \{\mathtt{VARIABLE\_DECLARATION\_STATEMENT}\} \wedge$ $PE \notin \{\mathtt{FOR\_INIT}\}$ | Data |
| $CT \in \{\mathtt{UNCLASSIFIED\_CHANGE}\} \wedge$ $CE \in \{\mathtt{MODIFIER}\}$ | Data |
| $CT \in \{\mathtt{STATEMENT\_*}\} \wedge$ $CE \in \{\mathtt{METHOD\_INVOCATION,\ CONSTRUCTOR\_INVOCATION,\ SYNCHRONIZED\_STATEMENT,\ CLASS\_INSTANCE\_CREATION}\}$ | Interface |
| $CT \in \{\mathtt{ADDING\_FUNCTIONALITY,\ REMOVING\_FUNCTIONALITY}\} \wedge$ $CE \in \{\mathtt{METHOD}\}$ | Interface |
| $CT \in \{\mathtt{UNCLASSIFIED\_CHANGE}\} \wedge$ $CE \in \{\mathtt{TYPE\_PARAMETER}\}$ | Interface |
| $CT \in \{\mathtt{STATEMENT\_*}\} \wedge$ $CE \in \{\mathtt{IF\_STATEMENT,\ FOREACH\_STATEMENT,\ CONTINUE\_STATEMENT,\ RETURN\_STATEMENT,\ THROW\_STATEMENT,\ SWITCH\_CASE,\ SWITCH\_STATEMENT,\ BREAK\_STATEMENT,\ CATCH\_CLAUSE,\ TRY\_STATEMENT,\ FOR\_STATEMENT,\ WHILE\_STATEMENT,\ DO\_STATEMENT,\ LABELED\_STATEMENT}\}$ | Logic/Control |
| $CT \in \{\mathtt{STATEMENT\_*}\} \wedge$ $CE \in \{\mathtt{ASSIGNMENT,\ POSTFIX\_EXPRESSION,\ PREFIX\_EXPRESSION}\} \wedge$ $PE \in \{\mathtt{FOR\_INCR}\}$ | Logic/Control |
| $CT \in \{\mathtt{STATEMENT\_*}\} \wedge$ $CE \in \{\mathtt{VARIABLE\_DECLARATION\_STATEMENT}\} \wedge$ $PE \in \{\mathtt{FOR\_INIT}\}$ | Logic/Control |
| $CT \in \{\mathtt{STATEMENT\_*}\} \wedge$ $CE \in \{\mathtt{ASSERT\_STATEMENT}\}$ | Other |

## Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2019.110421.

## References

Aickin, M., Gensler, H., 1996. Adjusting for multiple testing when reporting research results: the bonferroni vs holm methods.. Am. J. Public Health 86 (5), 726–728.

Ammann, P., Offutt, J., 2016. Introduction to Software Testing. Cambridge University Press.

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering. ACM, pp. 402–411.

Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S., 2006. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. 32 (8), 608–624.

Apache Software Foundation, 2017a. Maven FailSafe Plugin Website. http://maven.apache.org/surefire/maven-failsafe-plugin/. [accessed 30-November-2018].

Apache Software Foundation, 2017b. Maven Project Homepage. https://maven.apache.org/. [accessed 30-November-2018].

Apache Software Foundation, 2017c. Maven Surefire Plugin Website. http://maven.apache.org/surefire/maven-surefire-plugin/. [accessed 30-November-2018].

Apache Software Foundation, 2018. commons-io GitHub. https://github.com/apache/commons-io. [accessed 30-November-2018].

Athanasiou, D., Nugroho, A., Visser, J., Zaidman, A., 2014. Test code quality and its relation to issue handling performance. IEEE Trans. Softw. Eng. 40 (11), 1100–1125.

Beller, M., Gousios, G., Zaidman, A., 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: Proceedings of the 14th International Conference on Mining Software Repositories.

Beust, C., 2015. TestNG Documentation. http://testng.org/doc/. [accessed 30-November-2018].

Boehm, B.W., 1984. Verifying and validating software requirements and design specifications. IEEE Softw. 1 (1), 75.

Borges, H., Hora, A., Valente, M. T., 2017. [dataset] improved list of popular github repositories. https://doi.org/10.5281/zenodo.804473. [accessed 30-November-2018].

Borle, N.C., Feghhi, M., Stroulia, E., Greiner, R., Hindle, A., 2017. Analyzing the effects of test driven development in github. Empirical Softw. Eng. 1–28.

Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M., 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, pp. 597–608.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.-Y., 1992. Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on Software Engineering 18 (11), 943–956.

Cohn, M., 2010. Succeeding with agile: software development using Scrum. Pearson Education.

Coles, H., 2017. PIT Project Homepage. http://pitest.org/. [accessed 30-November-2018].

Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. Pit: a practical mutation testing tool for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 449–452.

Coplien, J. O., 2014a. Seque. http://rbcs-us.com/documents/Segue.pdf. [accessed 30-November-2018].

Coplien, J. O., 2014b. Why Most Unit Testing is Waste. http://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf. [accessed 30-November-2018].

Crispin, L., Gregory, J., 2009. Agile testing: A practical guide for testers and agile teams. Pearson Education.

Daran, M., Thévenod-Fosse, P., 1996. Software error analysis: A real case study involving real faults and mutations. In: ACM SIGSOFT Software Engineering Notes, 21. ACM, pp. 158–171.

Delamaro, M.E., Maidonado, J., Mathur, A.P., 2001. Interface mutation: an approach for integration testing. IEEE Trans. Softw. Eng. 27 (3), 228–247.

Demeyer, S., Ducasse, S., Nierstrasz, O., 2002. Object-oriented reengineering patterns. Elsevier.

Django Software Foundation, 2019. Django homepage. https://www.djangoproject.com/. [accessed 17-July-2019].
Dodds, K. C., 2017. Write tests. Not too many. Mostly integration.https://blog.kentcdodds.com/write-tests-not-too-many-mostly-integration-5e8c7fff591c. [accessed 30-November-2018].
Facebook Open Source, 2019. Graphql homepage. https://graphql.org/. [accessed 17-July-2019].
Fielding, R.T., Taylor, R.N., 2000. Architectural styles and the design of network-based software architectures, 7. University of California, Irvine Doctoral dissertation.
Fluri, B., Gall, H.C., 2006. Classifying change types for qualifying change couplings. In: 14th IEEE International Conference on Program Comprehension. IEEE, pp. 35–45.
Fraser, G., Zeller, A., 2011. Generating parameterized unit tests. In: Proceedings of the 20th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 364–374.
Gambi, A., Kappler, S., Lampel, J., Zeller, A., 2017. Cut: automatic unit testing in the cloud. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 364–367.
Garousi, V., Yildirim, E., 2018. Introducing automated gui testing and observing its benefits: an industrial case study in the context of law-practice management software. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 138–145.
Ghosh, S., Mathur, A.P., 2001. Interface mutation. Softw. Test., Verif. Reliab. 11 (4), 227–247.
Grechanik, M., Devanla, G., 2016. Mutation integration testing. In: IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 353–364.
Hayes, J.H., 2003. Building a requirement fault taxonomy: Experiences from a nasa verification and validation research project. In: 14th International Symposium on Software Reliability Engineering. IEEE, pp. 49–59.
Hayes, J.H., Raphael, C.I., Surisetty, V.K., Andrews, A., 2005. Fault links: exploring the relationship between module and fault types. In: European Dependable Computing Conference. Springer, pp. 415–434.
Holling, D., Hofbauer, A., Pretschner, A., Gemmar, M., 2016. Profiting from unit tests for integration testing. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 353–363.
IEEE, 2010. Systems and software engineering – vocabulary. ISO/IEC/IEEE 24765:2010(E) 1–418. doi:10.1109/IEEESTD.2010.5733835.
International Software Testing Qualification Board,. Int. Softw. Test. Qualification Board Glossary. http://www.astqb.org/glossary/search/unit. [accessed 30-November-2018].
Jenkins Contributors, 2018. Jenkins. https://jenkins.io/. [accessed 30-November-2018].
JUnit Team, 2017. JUnit Homepage. http://junit.org/junit5/. [accessed 30-November-2018].
Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. 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. ACM, pp. 654–665.
Kanstrén, T., 2008. Towards a deeper understanding of test coverage. J. Softw. 20 (1), 59–76.
Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., Le Traon, Y., 2017. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. Emp. Softw. Eng. 1–38.
Lachmann, R., Lity, S., Al-Hajjaji, M., Fürchtegott, F., Schaefer, I., 2016. Fine-grained test case prioritization for integration testing of delta-oriented software product lines. In: Proceedings of the 7th International Workshop on Feature-Oriented Software Development. ACM, pp. 1–10.
Laurent, T., Papadakis, M., Kintis, M., Henard, C., Le Traon, Y., Ventresque, A., 2017. Assessing and improving the mutation testing practice of pit. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 430–435.
Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. 50–60.
MVN Repository, 2018. MVN Repository - Most Popular. https://mvnrepository.com/popular. [accessed 30-November-2018].
Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing. John Wiley & Sons.
Namin, A.S., Kakarla, S., 2011. The use of mutation in testing experiments and its sensitivity to external threats. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, pp. 342–352.
Offutt, A.J., Hayes, J.H., 1996. A semantic model of program faults. In: ACM SIGSOFT Software Engineering Notes, 21. ACM, pp. 195–200.
Olkin, I., 1960. Contributions to probability and statistics: essays in honor of Harold Hotelling. Stanford University Press.
Oracle, 2017. What Is a Package?https://docs.oracle.com/javase/tutorial/java/concepts/package.html. [accessed 30-November-2018].
Orellana, G., Laghari, G., Murgia, A., Demeyer, S., 2017. On the differences between unit and integration testing in the travistorrent dataset. In: Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, pp. 451–454.
Papadakis, M., Henard, C., Harman, M., Jia, Y., Le Traon, Y., 2016. Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 354–365.
Papadakis, M., Jia, Y., Harman, M., Le Traon, Y., 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 936–946.
Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2017. Mutation testing advances: an analysis and survey. Adv. Comput..
Papadakis, M., Shin, D., Yoo, S., Bae, D.-H., 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In: 40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden.
Royce, W., 1999. Software project management. Pearson Education India.
Saff, D., Ernst, M.D., 2004. An experimental evaluation of continuous testing during development. In: ACM SIGSOFT Software Engineering Notes, 29. ACM, pp. 76–85.
Schuler, D., Zeller, A., 2013. Covering and uncovering equivalent mutants. Softw. Test. Verif. Reliab. 23 (5), 353–374.
SciPy developers, 2017. SciPy Website. https://www.scipy.org/. [accessed 30-November-2018].
Shapiro, S.S., Wilk, M.B., 1965. An analysis of variance test for normality (complete samples). Biometrika 52 (3/4), 591–611.
Spillner, A., Linz, T., Schaefer, H., 2014. Software testing foundations: a study guide for the certified tester exam. Rocky Nook, Inc..
Student, 1908. The probable error of a mean. Biometrika 6 (1), 1–25. doi:10.1093/biomet/6.1.1.
Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C., 2014. Bug characteristics in open source software. Emp. Softw. Eng. 19 (6), 1665–1705.
Trautsch, F., 2018. BugFixClassifier GitHub. https://github.com/ftrautsch/BugFixClassifier. [accessed 30-November-2018].
Trautsch, F., Grabowski, J., 2017. Are there any unit tests? an empirical study on unit testing in open source python projects. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 207–218.
Trautsch, F., Herbold, S., Grabowski, J., 2018. [dataset] Replication Kit. https://doi.org/10.5281/zenodo.2267946. [accessed 30-November-2018].
Trautsch, F., Herbold, S., Makedonski, P., Grabowski, J., 2016. Adressing problems with external validity of repository mining studies through a smart data platform. In: Proceedings of the 13th International Conference on Mining Software Repositories. ACM, pp. 97–108.
Travis CI GmbH, 2017. Travis CI Homepage. https://travis-ci.org/.[accessed 30-November-2018].
Tu, Q., et al., 2000. Evolution in open source software: A case study. In: Proceedings of the Internation Conference on Software Maintenance. IEEE, pp. 131–142.
Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2017. There and back again: can you compile that snapshot? J. Softw 29 (4).
Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M., 2007. On the detection of test smells: a metrics-based approach for general fixture and eager test. IEEE Trans. Softw. Eng. 33 (12), 800–817.
Wacker, M., 2015. Just say no to more end-to-end tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html. [accessed 17-July-2019].
Wang, R., Lagakos, S.W., Ware, J.H., Hunter, D.J., Drazen, J.M., 2007. Statistics in medicinereporting of subgroup analyses in clinical trials. New Engl. J. Med. 357 (21), 2189–2194.
Xia, X., Lo, D., Wang, X., Zhou, B., 2014. Automatic defect categorization based on fault triggering conditions. In: Proceedings of the 19th International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE, pp. 39–48.
Xu, D., Xu, W., Tu, M., Shen, N., Chu, W., Chang, C.-H., 2016. Automated integration testing using logical contracts. IEEE Transa. Reliab. 65 (3), 1205–1222.
You, E., 2019. Vue.js: The progressive javascript framework. https://vuejs.org/. [accessed 17-July-2019].
Zhao, Y., Leung, H., Yang, Y., Zhou, Y., Xu, B., 2017. Towards an understanding of change types in bug fixing code. Inf. Softw. Technol. 86, 37–53.

**Fabian Trautsch** is a PostDoc at the University of Goettingen. He received the BSc degree in applied computer science from the University of Goettingen in 2013, the subsequent MSc degree in 2015 and his doctorate in 2019. Since then he is working as a PostDoc in the Software Engineering for Distributed Systems group at the Institute of Computer Science of the University of Goettingen. His research interests include mining software repositories, software evolution, software testing, and empirical software engineering.

**Steffen Herbold** is a PostDoc and substitutional head of the research group Software Engineering for Distributed Systems at the Institute of Computer Science of the University of Goettingen. He received his doctorate in 2012 from the University of Goettingen. Dr. Herbolds research interests is empirical software engineering, especially the application of data science methods. His research includes work on defect prediction, repository mining infrastructures, usage-based testing, model-based testing, as well as scaling analysis in cloud environments.

**Jens Grabowski** is professor at the University of Goettingen. He is vice director of the Institute of Computer Science and is heading the Software Engineering for Distributed Systems Group at the Institute. Prof. Grabowski is one of the developers of the standardized testing languages TTCN-3 and UML Testing Profile. The current research interests of Prof. Grabowski are directed towards model-based development and testing, managed software evolution, and empirical software engineering.