Are there any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects

Fabian Trautsch, Jens Grabowski
Institute of Computer Science
Georg-August-University Göttingen
Göttingen, Germany
{trautsch,grabowski}@cs.uni-goettingen.de

***Abstract*—Unit testing is an essential practice in Extreme Programming (XP) and Test-driven Development (TDD) and used in many software lifecycle models. Additionally, a lot of literature deals with this topic. Therefore, it can be expected that it is widely used among developers. Despite its importance, there is no empirical study which investigates, whether unit tests are used by developers in real life projects at all.**

This paper presents such a study, where we collected and analyzed data from over 70K revisions of 10 different Python projects. Based on two different definitions of unit testing, we calculated the actual number of unit tests and compared it with the expected number (as inferred from the intentions of the developers), had a look at the mocking behavior of developers, and at the evolution of the number of unit tests.

Our main findings show, (i) that developers believe that they are developing more unit tests than they actually do, (ii) most projects have a very small amount of unit tests, (iii) developers make use of mocks, but these do not have a significant influence on the number of unit tests, (iv) four different patterns for the evolution of the number of unit tests could be detected, and (v) the used unit test definition has an influence on the results.

I. Introduction

Unit testing is an important development practice in, e.g., Extreme Programming (XP) [1] and Test-driven Development (TDD) [2]. Furthermore, unit testing is an essential phase in many software lifecycle models, e.g., in the V-model according to Boehm [3]. In this model, the component (or unit) test is the first test level, which verifies each software component [4].

It is believed, that unit testing is widely used because of its advantages: unit tests are easily automatable [2], they can serve as a source of documentation [5], help people to get into the project, and problems that are detected with a unit test can be directly traced back to the source [4].

Furthermore, a lot of scientific literature, which is focused on unit testing exists, e.g., on finding links between the test code and the code that is tested [6], [7], how to use mock objects [8], finding test smells [9], [10], [11], proposing test refactorings [10], [11], detecting test refactorings [10], [11], visualizing test executions [12], test case minimization [13], [14], and test generation [15], [16], [17].

978-1-5090-6031-3/17 $31.00

Even though we know about the importance of unit testing, we do not know whether unit tests are used by developers at all. Additionally, we do not know if developers use mocks to isolate the units they are testing. There is no empirical study which investigates, whether unit tests are used in real life projects. Hence, within this paper we present the results of a study on unit testing conducted on 10 open source software projects developed in Python. We decided on Python, as it gains more and more popularity in the development of software [18]. As basis for our study, we use the unit test definitions of the International Software Testing Qualification Board (ISTQB) [19] and the Institute of Electrical and Electronics Engineers (IEEE) [20]. To steer our research, we refine our goal into four different Research Questions (RQs):

RQ1: Are developers categorizing their tests into unit and non-unit tests correctly (using the IEEE and ISTQB definitions as basis)?

RQ2: How many tests in the examined projects can be categorized as unit tests?

RQ3: Do developers use mocks and how are they mocking? Does it influence the number of unit tests?

RQ4: How does the number of unit tests evolve over time?

In a practical context, the answers to these questions provide us with the opportunity to gain a deeper understanding of the current practice of testing in the real world. Furthermore, these answers could help to raise the sensitivity of practitioners and academia regarding the topic of unit testing, which could result in an improved education in the field of software testing and a higher quality of software.

In this paper we perform an exploratory study, in which we investigate the usage of unit tests in 10 open source software projects. Furthermore, we evaluate our findings with an statistically significant sample and give answers to all four RQs. Additionally, we provide a replication kit [21], including all data and implementations to ease replication and extension.

The remainder of this paper is structured as follows. In Section II the foundations for this paper are presented. Then, in Section III, we describe our study design, including the studied projects, the data collection and data analysis process,

our validation procedures, as well as our replication kit. The results of the study are reported in Section IV. For each RQ, the results are described separately. In Section V, we highlight different threats to validity of our study. Then, in Section VI, we compare our work with related work in this field. Finally, we conclude and describe future work in Section VII.

II. Background

This section presents the unit testing terminology, which is used along this paper. We present two definitions of units and unit testing, which are often used in the literature: one from the ISTQB [19], as well as one from the IEEE [20]. Moreover, we compare them with each other and highlight differences. As we are categorizing tests into unit and non-unit tests based on their imports, we explain the import system of Python in more detail. Finally, we apply the unit test definitions to Python.

A. Unit Testing

Different definitions of units and unit testing exist in the literature. In this paper, we focus on the definitions of unit and unit testing found in the ISTQB glossary [19] and in the IEEE 24765:2010 standard [20]. The ISTQB defines a unit as follows [19]:

> *Definition 1 (ISTQB Unit):* A minimal software item that can be tested in isolation.

The term "minimal software item" is not further defined. Therefore, we define "minimal software item" **as the smallest compilable unit**, as this is the smallest software item that can work independently (e.g., a class in Java). Furthermore, this definition is used in other publications (e.g., [22]). Definition 1 highlights that it should be possible to test the unit in isolation. This can be accomplished by using mocks [8] for the unit's dependencies, that should not be tested. The unit under test then utilizes the mocks functionality as if it was the real dependency. The IEEE defines a unit as follows [20]:

> *Definition 2 (IEEE 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.
> 4) a quantity adopted as a standard of measurement.

In the ISTQB definition the term unit has two synonyms: module and component, whereas the IEEE states that the relationship between the terms "module", "component", and "unit" is not yet standardized [20]. Definition 2 directly states, that a unit can not be subdivided into other components. This is similar to the "minimal software item" in Definition 1. Therefore, both definitions define a unit as a software item, that can not be further subdivided. Furthermore, the item must be logically separable from the program, as an isolation can not be achieved otherwise.

For our purpose, we also need to define unit testing. Based on the definitions of units, both the ISTQB and IEEE define unit testing as follows:

> *Definition 3 (ISTQB Unit Testing):* The testing of individual software components.

> *Definition 4 (IEEE Unit Test):*
> 1) testing of individual routines and modules by the developer or an independent tester.
> 2) a test of individual programs or modules in order to ensure that there are no analysis or programming errors.
> 3) test of individual hardware or software units or groups of related units

Hence, for the ISTQB a unit test only considers a single minimal software item, while the IEEE definition allows testing of *groups* of related units. Note, that if a test is a unit test in respect to the ISTQB definition it is also a unit test in terms of the IEEE definition.

The testing is done by implementing tests. Tests are defined as "A set of one or more test cases." [23], whereas a test case is "A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement." [24]. The definitions of the IEEE are similar to the ones cited here.

In the following, we will use the term "ISTQB unit test" for a test that is a unit test in respect to Definition 3 and "IEEE unit test" for a test that is a unit test in terms of Definition 4.

B. Python Import System

Python has a very flexible import system. Different import statements can result in the same import behavior. In Python it is even possible to perform imports in functions [25].

Python has the concept of *packages*, which helps to organize modules and provides a naming hierarchy. Packages can be seen as directories on the file system, whereas *modules* are the files within these directories [25]. Furthermore, it is possible that a package has several sub-packages. Another important aspect are the *__init__.py* files. A directory is only recognized as package, if such a file is placed in it. Then, if a module of the package is imported, the *__init__.py* file (which can contain code, further imports, etc.) is implicitly executed. Figure 1 shows an example structure of a project.

If *locations.py* imports the module *download.py* the *__init__.py* files of the packages *pip* and *commands* are implicitly executed. Therefore, the imported modules are: *pip/commands/download.py*, *pip/__init__.py*, and *pip/commands/__init__.py*.

C. Unit Testing in Python

To address our RQs stated in Section I, we need to apply Definitions 1-4 to Python projects. There is no standard, which

```
pip/
    __init__.py
    commands/
        __init__.py
        download.py
    location/
        __init__.py
        locations.py
```

Fig. 1. Example project structure, showing packages and its modules.

regulates how Python projects should be structured. Most projects have a separated test folder, but this is not compulsory. In Python, the smallest compileable unit is a module. It can be logically separated from the program and contains all the logic needed for one part of the program [26]. Classes cannot be seen as units, as a program in Python can be written completely without any class, which is, e.g., different to Java.

In Python, tests are subdivided into different modules that can contain multiple test cases. If we take the definitions above into account, an ISTQB unit test in Python tests only one module. As in Python related modules are put into the same package, a test is an IEEE unit test, if it tests one module or a group of modules, which originate from the same package. Note, that ISTQB unit tests are a subset of IEEE unit tests.

III. Empirical Study Design

This section describes the study design in more detail. We have taken the guidelines by Runeson et al. [27] as a basis for the reporting of our case study. They proposed to divide the reporting of the case study design into five different parts: research questions, case and subject selection, data collection procedures, analysis procedures, and validation procedures. We have exchanged the subsection regarding the RQs (as these were stated in Section I, which also describes the case selection), with a subsection describing our replication kit, as we believe that replicating research is essential and especially important to empirical software engineering research [28], [29].

Figure 2 gives an overview of our approach. The data collection process comprises of two steps: first (*Step 1*), the whole revision history of the project is collected from its git repository and stored in a MongoDB [30]. In *Step 2* each revision of the project is processed. For each revision, all test imports are detected, as well as mock usage and mocked imports. The data is then linked to the data of *Step 1* and stored in the same MongoDB.

The data analysis process is divided into two steps: first the collected data is read from the MongoDB. Now each test of each revision of the project is put into different categories (*Step 3*). Afterwards, a Comma-Separated Values (CSV) file is generated for the project, where the number of category members for each revision is listed. As second analysis step (*Step 4*), the CSV file is read and an analysis in respect to the different RQs formulated in Section I is performed. More details regarding the different steps of our approach are given in the following subsections.

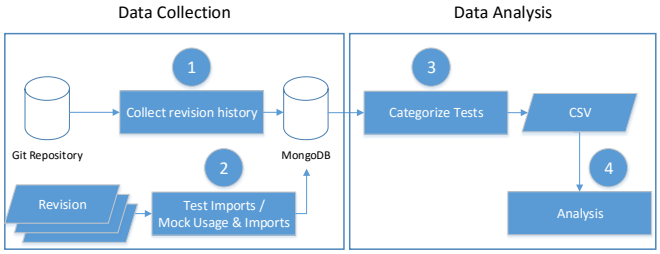

Fig. 2. Overview of our approach, which can be separated into data collection and data analysis. Rectangles depict processes (steps), whereas parallelograms depict in- and output data.

A. Studied Projects

We selected our studied projects randomly from GitHub [31] via the *explore* function and from a list of Python projects located at GitHub [32]. We defined the following inclusion criteria to which the projects must comply, as the executed analysis would be futile or not possible otherwise.

- The project shall have tests.
- The project shall be written in Python.
- The project shall have more than 200 revisions.
- The tests of the projects shall be separated in unit and other tests (e.g., located in different folders).

Furthermore, to reduce the domain bias, we selected projects from different application areas (e.g., machine learning, package managers, testing). Additionally, we have also included projects that are mainly industrial driven (i.e., aws-cli, nupic, and ansible) to extend our focus. Table I reports the chosen projects together with several project related characteristics. We excluded 2,808 revisions, which were not done on the main branch of the projects, i.e., the *master* branch[1]. The reason for this is that we want to reduce the noise in the data, which stale branches introduce.

Furthermore, Table I reports the number of tests for each project. This number is calculated by summing up the number of tests of each revision.

The number of file changes represent how often files were changed in the project, which is an indicator for the activity and size of the project. The data time span shows the dates of the first and last revision, which were analyzed. Furthermore, Table I highlights the main application field in which the project is used. Additionally, it lists if the projects are completely community-driven or if there is industrial support of the project. This is important to get a better understanding of these projects, as open source does not necessarily mean that there is no company supporting the project. This can also have an impact on the results, as a company might enforce certain standards or a strict software development process [33].

B. Data Collection

In the following, we explain the two data collection steps of our approach in more detail. For each of the data collection steps, we answer the questions: what data is collected, why

[1]Please note, that the project ansible does not have a branch called "master". But the main working branch for this project is the "devel" branch. Hence, we have chosen this branch as the main branch for ansible.

TABLE I
OVERVIEW OF THE ANALYZED PROJECT DATA.

Project	#Revisions	#Tests	#File Changes	Data Time Span	Application Field	Main Project Driver
ansible [34]	19,992	396,857	49,514	2012-02 - 2016-07	DevOps	Ansible Inc.
aws-cli [35]	3,958	476,644	42,042	2012-11 - 2016-07	Cloud Computing	Amazon Web Services Inc.
nose [36]	1004	75,087	3726	2006-12 - 2016-03	Testing	Community
nose2 [37]	784	29,294	2,764	2010-08 - 2016-06	Testing	Community
nupic [38]	5,825	549,029	346,371	2013-04 - 2016-07	Machine Learning	Numenta
pip [39]	4,745	161,506	16,661	2008-10 - 2016-07	Package Manager	Community
robotframework [40]	11,083	823,056	39,150	2008-05 - 2016-07	Testing	Community
scikit-learn [41]	20,969	2,025,313	105,271	2010-01 - 2016-07	Machine Learning	Community
vcrpy [42]	657	12,520	1,503	2012-05 - 2016-07	Virtual Reality	Community
warehouse [43]	1,937	87,926	6,872	2015-01 - 2016-07	Package Manager	Community
Overall:	**70,953**	**4,637,232**	**613,874**	**2006-12 - 2016-07**	**6** different fields	**3** industrial, **7** community

(for which purpose), and how. We used the new version of our SmartSHARK platform [44] for the data collection.

Step 1: Collect revision history

What? First, we collect the whole revision history of a project, including all commits, branches, file changes, and differences between file versions. Out of this data, we can reconstruct the whole project structure at each revision.

Why? This step provides us with the needed meta-data from the project for the next steps and is used in *Step 3* to determine, which tests are categorized as unit test by the developers.

How? We use our tool called *vcsSHARK* [45]. The vcsSHARK is using the official git library [46] to parse all commits of a project. The data that are gathered in this process is transformed and a model is built from it. This model is then stored via the mongoengine [47] document-object mapper in a MongoDB.

Step 2: Collect test imports and mocked imports for each revision

What? In this step, we collect all imports for each test state. Furthermore, we collect data about the mock usage of a test state and its mocked imports.

Why? As noted in Section II-C, in Python a module is considered a unit. Therefore, we need to collect information about the module imports of a test. The rationale behind this is that all modules, which are tested, must be imported by the test. Therefore, if a test has only one import it can be considered as an ISTQB unit test. Otherwise, if all imports of a test belong to the same package, it can be classified as an IEEE unit test.

Furthermore, we need to collect data about the mock usage in tests and identify mocked imports to analyze, if mocks are used by developers and if the usage of mocks has an influence on the categorization stated above.

How?

To perform the data collection in this step, we use our tool called *testImpSHARK* [48]. It first checks out a given revision to analyze it[2]. Then, it runs its test detection procedure, which detects Python files that have *test* in their file name. Finally, it tries to detect the imports and runs through the following steps for each detected test:

1) **detection of mock usage via regular expressions** (used on source code): check if a test uses mocks by detecting if a module with the word *mock* in it is imported (e.g., the python standard mocking library [49]).
2) **detection of mocked imports via regular expressions** (used on source code): if a test uses mocks, the program detects the mocked imports by checking which class or function is mocked and looks up the corresponding module. We classify an import as mocked, if at least one function, class, or method of this import is mocked. The used regular expressions are reported in Table II.
3) **get direct imports**: the program detects all direct imports of the test using the modulegraph [50] library.
4) **get all imports recursively**: the program detects all imports recursively using the modulefinder [51] module of the Python standard library.
5) **get mock-cutoff imports**: the program detects all imports recursively, where modules that are mocked, and their imports, are not considered any further.

At each step, we filter out imports, that are not part of the project, e.g. standard libraries, as we only want to analyze the projects contents. Finally, a model of the data is built and the newly collected data is linked with the data acquired in *Step 1* and stored in the same MongoDB.

C. Data Analysis

In the following, the two data analysis steps, which correspond to *Step 3* and *Step 4* of our approach (see: Figure 2), are explained in more detail. For each of the data analysis steps, we answer the questions: what is done in the analysis, why, and how.

Step 3: Categorize Tests

What? We use the data collected in *Step 1* and *Step 2* by accessing the MongoDB. In this step, we put all tests (with at least one import) of all revisions of the project into different categories. The rows of the resulting CSV file are then the counts of the number tests in each category for each revision.

Why? The categorization of the tests is done, as this information is needed to answer the RQs formulated in Section I.

The reason for the creation of a CSV file is two-fold: first, we want to summarize the category counts on revision level and not on, e.g., project level, as we have a more detailed

[2]The testImpSHARK can analyze projects that are compatible to both major versions of Python (2.7, 3.5).

TABLE II
REGULAR EXPRESSIONS USED TO DETECT MOCKS.

Regular Expression	Description
`\s*(?:@patch\.object|patch\.object)\s*\(\s*([\w\.]*)\s*\,`	Detect mocked classes
`\s*(?:@patch|mock\.patch|patch)\s*\(\s*(?:\'|\")\s*([\w\.]*)`	Detect mocked functions/methods

view on the data this way and we are able to detect changes of the counts over time. Second, the resulting CSV file can be analyzed by a number of tools (e.g., R or Python).

How? For the categorization of the tests, we first analyze all tests of the project, that were collected by our tool (see: Section III-B). Note, that we skip tests, where the number of imports is 0. This can happen, as our tool may find tests, which are not real tests (e.g., they just contain data for the test) or when the data collection tool has thrown an error (see: Section V). The following steps are executed for each detected test:

1) **discard imports:** a filter is applied to the collected imports of the test, i.e., the direct imports, recursively collected imports, mocked imports, and mock-cutoff imports. The filter discards imports, which contain only data or configurations for the test, e.g., database models, configurations, __init__.py files, or other tests. Furthermore, we assume that everything which is inside the test folder is test data (e.g., a test class from which all others inherit) or other tests. These imports are discarded. This filter ensures, that we do not categorize tests as non-unit tests, because they are importing modules, which do not contain any testable logic (e.g., configurations). The *remaining imports* are used for the categorization.
   The filter criteria was determined by analyzing the modules of the projects and their contents and tested with manually-curated data samples.
2) **test categorization:** after the filter is applied, we put the tests into one (or more) categories. We have created categories for tests that are ISTQB unit tests (*istqb*), IEEE unit tests (*ieee*), and developer unit tests (*dev*). We define the term "developer unit test" as a test, where the developers think it is a unit test. Furthermore, there are categories to analyze the mocking behavior of developers: "WM" (*istqbWM*, *ieeeWM*) for tests, which are ISTQB or IEEE unit tests, if we exclude the mocked imports and "MC" (*istqbMC*, *ieeeMC*) for tests, which are ISTQB or IEEE unit tests, if we only use the mock-cutoff imports defined in Section III-B. The rationale behind these mocking categories is that we want to implement a pessimistic view on mocking (WM categories) and an optimistic view (MC categories). Pessimistic means that we assume that only the mocked imports are marked as mocked, but not their imports. Optimistic means that we assume, that the developer took care of all the imports of the module, that is mocked in the test, i.e., these imports should not be taken into account for the categorization.

After the categorization of all tests for the processed revision, the number of tests in these categories are calculated and a row in the CSV file is created.

TABLE III
DESCRIPTION OF THE DIFFERENT CATEGORIES USED IN THE CATEGORIZATION STEP.

Category	Included Tests
all	Tests that have at least one dependency.
dev	Tests that are developer unit tests. This is determined by looking at the path of the test. If the path contains folders like "utest" or "unit" we assume that the developer classified this test as unit test. The project scikit-learn is one special case, as the unit tests are "located in test subdirectories"[52], as stated in the official contribution guide [52].
istqb	Tests that are ISTQB unit tests, i.e., they only import one module.
ieee	Tests that are IEEE unit tests, i.e., all imports of the test belong to the same package.
istqbD	Intersection of the categories *istqb* and *dev*.
ieeeD	Intersection of the categories *ieee* and *dev*.
istqbWM	Tests that are ISTQB unit tests, if we exclude mocked imports.
ieeeWM	Tests that are IEEE unit tests, if we exclude mocked imports.
istqbMC	Tests that are ISTQB unit tests, if we only include mock-cutoff imports (see: Section III-B).
ieeeMC	Tests that are IEEE unit tests, if we only include mock-cutoff imports (see: Section III-B).

Step 4: Analysis

What? In this step, the projects are further analyzed. The analysis is done for each RQ separately.

Why? Out of the information we gain during the analysis, we construct the answers for our RQs formulated in Section I.

How? In the following, the "how?" is answered for each RQ individually.

For answering **RQ1**, we go through all projects (CSV-files). Only revisions, where the count for the category *dev* is greater than 0 are taken into account, i.e., at least one test is categorized as unit test by the developers. To answer **RQ1**, we need to assess, if there is a difference between the number of developer unit tests and the number of ISTQB/IEEE unit tests as determined by our tool in *Step 3*. For this, we are calculating the mean difference between the categories *dev* & *istqbD*, as well as *dev* & *ieeeD* over all revisions. Note, that we are using the categories *istqbD* and *ieeeD*, as we want to know how many of the tests in the category *dev* are real unit tests. The resulting mean differences then show how many tests are wrongly categorized by the developers on average per revision. Furthermore, we check for how many revisions the expected number of unit tests (as inferred from the developers' point of view) matches the actual number of unit tests (using Definition 3 and 4) and for how many it deviates for ≤ 1 and

≤ 5 tests. Furthermore, we create a boxplot of the different categories over all analyzed revisions to visualize the data.

Instead of focusing on revisions, where we have data about the developer's categorization of the test, we look at all revisions of the projects that have tests to answer **RQ2**. Therefore, we calculate the *mean* percentages to summarize how many tests (of *all* tests in the project) are put into the categories *ieee* or *istqb* and can therefore be seen as unit tests.

To answer **RQ3**, we first determine the usage of mocks in the different tests. Afterwards, we separate the tests into tests, that use MagicMocks (mocks, that are dynamically created and do not mock any import [49]) and tests that mock imports (non-MagicMocks). Then, we are calculating the average percentage of tests over all revisions, that use MagicMocks or non-MagicMocks. To check if non-MagicMocks have an influence on the categorization of tests, we create a subset of the data, where we only consider revisions, where at least one test is mocking an import. Furthermore, we calculate the differences between the categories *istqb* & *istqbWM*, *istqb* & *istqbMC*, *ieee* & *ieeeWM*, and *ieee* & *ieeeMC*. Hence, we can determine if there is a difference in the number of unit tests at a revision if we consider mocks in our analysis.

To visualize the evolutionary trend for **RQ4**, we analyze all revisions of the projects and visualize the counts for the categories *all*, *dev*, *ieee*, and *istqb*. This way, we can see if the number of unit tests are increasing, decreasing, or if another pattern can be identified. As we are including the *dev* category, we can also see how the number of unit tests is evolving over time in the eyes of the developers.

D. Validation Procedures

The first quality control mechanism that we employed to ensure the validity of our data collection data is the testing of our programs. We wrote tests for each program and checked them with manually-curated samples of the data to ensure that they are working as expected.

Furthermore, to ensure that our categorization and filtering is working correctly, we manually examined a data sample of tests, which were used for the analysis. For this we used a technique called *stratified sampling* [53]. Hence, we break our tests into groups (called strata) and select a random sample from each of these groups. This way, we ensure that we have a more balanced sample, than just drawing our tests randomly from the whole population.

We decided to use three stratas: tests which were categorized as 1) *ieee*, 2) *istqb*, and 3) nothing of both. We did not use all categories, as the resulting sample would be too large for a manual analysis. To apply stratified sampling, we first need to calculate the sample size for each of the stratas. We decided to use a confidence interval of length 10% within a 95% confidence level, taking into account our large population size. Hence, the sample size for each strata is calculated as follows [53]:

$$sample\ size = \frac{x}{1 + \frac{x-1}{\#tests\ in\ strata}}$$

with

$$x = \frac{Z^2 * p * (1-p)}{0.1^2}$$
$$Z = 1.96\ for\ 95\%\ conf.\ level$$
$$p = 0.5\ for\ population\ with\ unknown\ variability$$

Then, we check the correctness of all **randomly** drawn tests by manually categorizing it and comparing this categorization with the one that was automatically calculated.

The calculated sample size for our staratas are: 97 (ieee), 96 (istqb), and 97 (nothing of both). Hence, we manually validated 290 tests. From these, 277 were correct in terms of their categorization. Therefore, 95,52% of the sampled tests were correct. As we are using a confidence interval of length 10%, we can conclude with a confidence of 95%, that at least 85,52% of all tests are correctly categorized. The most common reason for the misscategorization was filtering out the unit under test.

E. Replication Kit

All the data used in our study are publicly available. Furthermore, we provide all the implementations that we have used to extract and analyze the data. Moreover, we provide the implementations that we have used to create the different plots in this paper. The replication kit can be found online [21]. We erased the collection where the developer names and email addresses are stored to comply with data privacy regulations.

IV. Study Results

In this Section, we present the results and provide answers for each RQ separately. We also discuss further findings in Section IV-E.

A. RQ1: Are developers categorizing their tests into unit and non-unit tests correctly (using the IEEE and ISTQB definitions as basis)?

Figure 3 shows boxplots for all examined projects. For each project, three different boxes are plotted for the categories *dev*, *istqbD*, and *ieeeD*. They show different distribution characteristics like the median, lower quartile, or upper quartile of the category for the corresponding project. The *dev* box of the project aws-cli has the highest median. Hence, the developers of this project believe that a revision in their project has about 100 unit tests. But the real value of ISTQB/IEEE unit tests is at about 5 unit tests per revision. The same trend can be observed for other projects.

Table IV shows the mean difference of the number of tests in the categories *dev* & *istqbD* and *dev* & *ieeeD* over all revisions. Furthermore, it depicts the mean percentage of unit tests per revisions that are developer unit test *and* ISTQB/IEEE unit tests. The calculated difference is low for each project with one exception, which is the project aws-cli. Hence, in this project there are more developer unit tests, which are also IEEE unit tests than developer unit tests which are also ISTQB unit tests. The calculated percentages highlight that just a small fraction

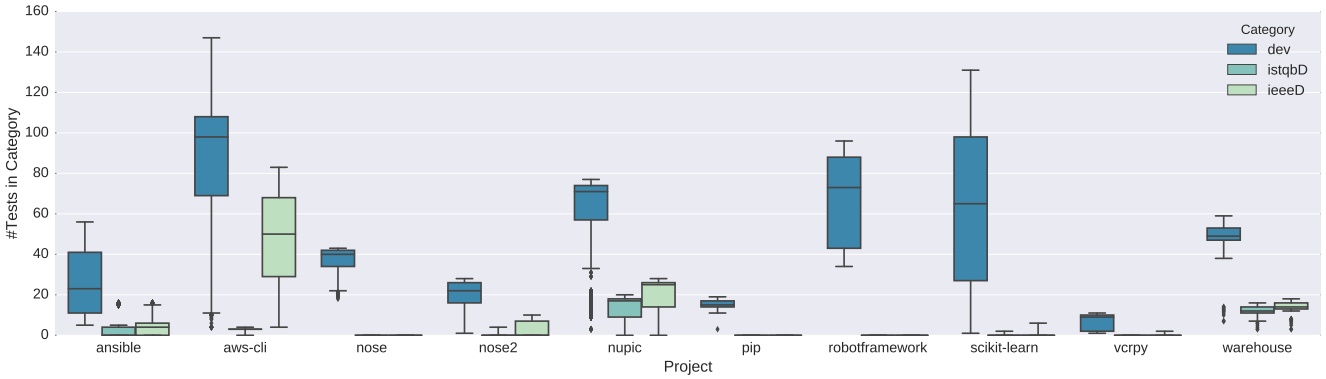

Fig. 3. Boxplot of the categories *dev*, *istqbD*, and *ieeeD* for the different projects.

TABLE IV
MEAN DIFFERENCE OF THE NUMBER OF TESTS IN THE CATEGORIES *dev* & *istqbD* AND *dev* & *ieeeD* OVER ALL REVISIONS. *istqbD%* AND *ieeeD%* DEPICTS THE MEAN PERCENTAGE OF UNIT TESTS PER REVISION THAT ARE DEVELOPER UNIT TESTS AND ISTQB/IEEE UNIT TESTS.

	Mean Difference		Percentages	
Projects	**dev&istqbD**	**dev&ieeeD**	**istqbD%**	**ieeeD%**
ansible	23.08	21.43	4.48%	20.61%
aws-cli	88.29	41.57	3.19%	60.16%
nose	36.81	36.81	0%	0%
nose2	20.32	17.50	5.73%	22.88%
nupic	46.20	40.37	22.04%	31.41%
pip	15.17	15.17	0%	0%
robotframework	66.15	66.15	0%	0%
scikit-learn	63.30	63.16	0.14%	2.56%
vcrpy	7.22	7.71	0%	9.86%
warehouse	36.77	34.88	25.49%	29.45%

or even *none* of the developer unit tests are also actual unit tests in respect to the IEEE/ISTQB definitions.

To quantify our results, Table V reports for how many revisions the number of developer unit tests and ISTQB/IEEE unit tests match (=0). Furthermore, it shows for how many revisions these numbers deviate for less than one test ($\leq$1) and less than five tests ($\leq$5). Note, that the higher these numbers are, the better it is as there is less deviation from the actual number of unit tests. Table V also highlights, the number of revisions that are analyzed for each project. It shows, that the number of developer unit tests deviates (for most revisions) for more than 5 tests from the number of ISTQB/IEEE unit tests.

The tables IV and V show a trend, which is confirmed by the boxplot in Figure 3. This leads to the following answer for our **RQ1**:

> **Answer to RQ1:** Our results show that the developers of the projects which we have examined, are not categorizing their tests correctly. The results highlight that there is a large deviation between the number of developer unit tests and the number of real unit tests (ISTQB/IEEE unit tests). Furthermore, the results show that it matters which unit test definition is used in this case.

B. RQ2: How many tests in the examined Projects can be categorized as Unit Tests?

Table VI reports the average percentage of ISTQB/IEEE unit tests in one revisions. The table shows, that there is a large variance between the projects in terms of their number of unit tests. Pip, e.g., does not have any ISTQB unit tests, whereas a revision in the project warehouse has about 26.35% ISTQB unit tests. The highest value has the project aws-cli with 68,30% of IEEE unit tests on average. This table leads to the following answer for **RQ2**:

> **Answer to RQ2:** The number of tests that can be categorized as unit tests depends on the project and on the unit test definition used. The values are ranging from 0% to 68.30%. The average percentage of IEEE unit tests in a revision is higher for all projects, as ISTQB unit tests are a subset of IEEE unit tests (see: Section II). One exception is the project robotframework, where both values are the same.

C. RQ3: Do developers use Mocks and how are they mocking? Does it influence the Number of Unit Tests?

Table VII reports the average percentage of tests in a revision that use MagicMocks and non-MagicMocks. This table shows, that the mock usage of developers differs from project to project. Warehouse, e.g., does not use any mocks, whereas aws-cli makes heavy use of MagicMocks. Furthermore, the percentage of tests that use MagicMocks in a revision is higher than the percentage of non-MagicMocks for each project.

To investigate whether non-MagicMocks have an influence on the number of unit tests in a revision, we investigated how this number changes if we take mocks into account for the categorization of the tests. For this, we had a look at the mean differences between the percentages of tests of the categories *istqb* & *istqbWM*, *istqb* & *istqbMC*, *ieee* & *ieeeWM*, and *ieee* & *ieeeMC*. From the results we generated Table VIII, which reports how the number of unit tests are changing, if we take mocks into account. This table highlights, that the number of ISTQB/IEEE unit tests are decreasing, when we exclude mocked imports (**WM** categories) or use our mock-cutoff (**MC** categories, see: Section III-B). E.g., for the project nupic the average number of unit tests is decreasing by over 20%. Hence,

TABLE V
NUMBER OF REVISIONS, WHERE THE DEVELOPERS CLASSIFICATION FIT OR DEVIATES FROM THE CALCULATED VALUE USING DEFINITION 3 AND 4. THE NUMBERS IN BRACKETS DEPICT THE PERCENTAGES OF REVISIONS THAT HAVE THE DEPICTED DEVIATION IN THEIR NUMBER OF UNIT TESTS. THE HIGHER THESE NUMBERS, THE BETTER.

		#Revisions, with different deviations					
Project	**#Revisions**	**=0 (istqb)**	**=0 (ieee)**	**≤1 (istqb)**	**≤1 (ieee)**	**≤5 (istqb)**	**≤5 (ieee)**
ansible	10,988	0	0	0	0	95 (0.86%)	3,403 (30.97%)
aws-cli	3,916	0	363 (9.27%)	0	398 (10.16%)	10 (0.26%)	423 (10.80%)
nose	1,003	0	0	0	0	0	0
nose2	684	11 (1.61%)	16 (2.34%)	25 (3.65%)	25 (3.65%)	25 (3.65%)	55 (8.04%)
nupic	5,822	0	0	0	0	9 (0.15%)	9 (0.15%)
pip	2,848	0	0	0	0	1 (0.04%)	1 (0.04%)
robotframework	11,000	0	0	0	0	0	0
scikit-learn	20,539	0	36 (0.18%)	49 (0.24%)	484 (2.36%)	1,195 (5.82%)	1,440 (7.01%)
vcrpy	558	0	55 (9.86%)	29 (5.20%)	55 (9.86%)	174 (31.18%)	174 (31.18%)
warehouse	1,569	0	0	0	0	2 (0.13%)	8 (0.51%)

TABLE VI
AVERAGE PERCENTAGE OF UNIT TESTS IN ONE REVISION USING DEFINITION 3 (ISTQB) AND 4 (IEEE).

Projects	**istqb %**	**ieee %**
ansible	3.25%	22.27%
aws-cli	2.61%	68.30%
nose	0.04%	0.04%
nose2	6.36%	24.78%
nupic	20.52%	28.03%
pip	0.00%	6.71%
robotframework	0.01%	0.01%
scikit-learn	0.84%	3.91%
vcrpy	0.00%	20.41%
warehouse	26.35%	30.69%

TABLE VII
AVERAGE PERCENTAGE OF TESTS THAT USE MAGICMOCKS AND THAT MOCK IMPORTS.

Project	**Use MagicMocks %**	**non-MagicMocks %**
ansible	21.07%	7.47%
aws-cli	89.23%	10.21%
nose	14.11%	0%
nose2	0.19%	0%
nupic	12.43%	7.99%
pip	26.97%	14.16%
robotframework	0.0003%	0%
scikit-learn	0.66%	0%
vcrpy	49.26%	6.42%
warehouse	0%	0%

over 20% of ISTQB/IEEE unit tests are no longer recognized as unit tests, if we take mocks into account.

Upon further investigation we found, that tests often mock parts of the unit under test (e.g., functions), which is treated by us as if the whole module is mocked. Therefore, a test previously correctly recognized as unit test is not categorized as unit test anymore, as the only import (the unit under test) is marked as mocked. As we are operating on module level to differentiate unit and non-unit tests from each other, we are not taking functions into account. Hence, we can not automatically detect the tests, which are mocking parts of their unit under test.

TABLE VIII
MEAN DIFFERENCES BETWEEN THE PERCENTAGES OF TESTS OF THE CATEGORIES *istqb* & *istqbWM* (ISTQBWM), *istqb* & *istqbMC* (ISTQBMC), *ieee* & *ieeeWM* (IEEEWM), AND *ieee* & *ieeeMC* (IEEEMC). PROJECTS THAT ARE NOT LISTED HERE DO NOT USE ANY NON-MAGICMOCKS.

Project	**istqbWM**	**ieeeWM**	**istqbMC**	**ieeeMC**
ansible	-4.30%	-5.00%	-3.28%	-3.82%
aws-cli	±0%	-2.49%	+1.11%	-62.22%
nupic	-20.52%	-28.03%	-20.52%	-28.03%
pip	±0%	+0.35%	±0%	-4.13%
vcrpy	±0%	±0%	±0%	±0%

Answer to RQ3: Our findings show that the mock usage depends on the project. There are projects, which heavily use mocks, but also projects that do not use mocks at all. But, MagicMocks are more often used, than non-MagicMocks. Surprisingly, we get less unit tests in most cases, if we exclude mocked imports or use our mock-cutoff.

D. RQ4: How does the Number of Unit Tests evolve over Time?

Table IX reports different patterns regarding the evolution of the number of ISTQB/IEEE unit tests. The first pattern that can be seen is an increase of unit tests in the beginning and then a stagnation at some point. For the project aws-cli, this behavior is only visible for ISTQB unit tests. The second observable pattern is a raise of unit tests in the beginning and a drop afterwards. For the project nose the number of unit tests is raising just the first few revisions and then dropping low. This is different for the project nose2, where the raise and the drop in unit tests is time-shifted for ISTQB and IEEE unit tests. Another pattern that we observe is, that the number of unit tests is constantly raising. The only project where this pattern is found is aws-cli, if we look at the evolution of the number of IEEE unit tests. Furthermore, another observable pattern is, that the number of unit tests is constantly low, which is the case for the project robotframework and pip (for ISTQB unit tests). For the project ansible, there is no clear pattern detectable.

TABLE IX
DETECTED PATTERNS FOR THE EVOLUTION OF THE NUMBER OF UNIT TESTS FOR THE EXAMINED PROJECTS.

Pattern	Projects
beginning raising → stagnation	warehouse, nupic, aws-cli (istqb)
beginning raising → drop	vcrpy, nose, scikit-learn, nose2, pip (ieee)
constantly raising	aws-cli (ieee)
constantly low	pip (istqb), robotframework
no clear pattern	ansible

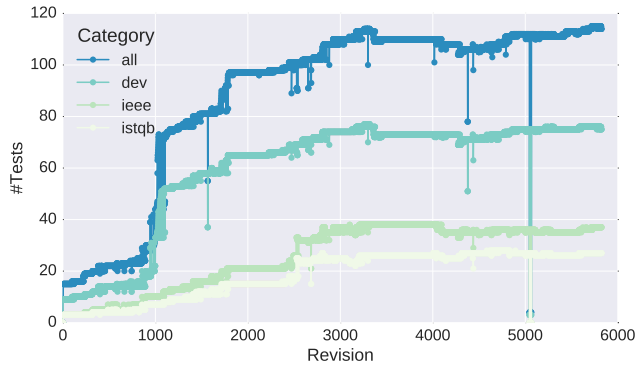

Fig. 4. Evolution of the number of tests in the categories *all*, *dev*, *ieee*, and *istqb* for the project nupic.

The evolution of the number of unit tests for the project nupic is shown in Figure 4. We can not show all graphs for all projects here, because of the space limitations. Nevertheless, all implementations to create these graphs are available in the replication kit (see: Section III-E). Figure 4 displays the counts of the categories *all*, *dev*, *ieee*, and *istqb* for the project nupic over time. We see a raise in the number of unit tests for this project till revision ∼1,800. Then the number stagnates and begins to raise with revision ∼2,500. Furthermore, we see a large drop at revision ∼5,000, as well as some smaller drops at different revisions. We checked these drops manually and found, that these are "wrongly placed" commits. Meaning, that a developer committed an older version of the project at this point in time. These drops do not influence our results for the other RQs, as we only look at differences between the counts of categories and the time of the revision do not influence these.

For most of the projects, the number of unit tests is rising in the beginning, which can be explained by the size of the project: in the beginning, few files are present in the project and therefore, a test is not able to have a lot of dependencies/imports. But the more complex a project grows, the more files are created and therefore the dependencies between these files are raising. The observed patterns lead to the following answer to our **RQ4**:

Answer to RQ4: We detected four different patterns for the examined projects as described in Table IX. For one project, no clear pattern was observable. We have seen that the evolution of the number of unit tests depends on the definition of unit testing used for two of the projects.

E. Further Findings

During the validation of our approach, we found that some developer unit tests, have only one *direct* import (i.e., the test directly imports only one other module). Hence, we hypothesize that developers do not look at the imports of the directly imported modules. Therefore, developers think that they are developing a unit test, but in fact several other modules are imported through this module. We did not look deeper into this hypothesis, but will do in future work.

V. THREATS TO VALIDITY

In this section, we want to discuss the threats to validity regarding our empirical study. We discuss the external, internal, and construct validity of our study.

External Validity. Threats to the external validity are concerned with the ability to generalize the results. In our study, we only had a look at Python projects. The results could be different for, e.g., Java projects. Although we have chosen projects, which have a wide variety in their characteristics (e.g., number of revisions, application field, industrial or community driven), the results can vary if other projects are chosen. Replication of this work using other Python projects (and projects using other programming languages) is required in order to reach a more general conclusion.

Internal Validity. Threats to the internal validity are concerned with the ability to draw conclusions from the relation between causes and effects. We are studying a manual classification of tests by developers in **RQ1**. There might be, by the nature of it, some misclassifications made by them.

Construct Validity. Construct threats to validity are concerned with the degree to which our analysis really analyses what we are claiming it does. We developed different tools for the data extraction and analysis process. We carefully tested our tools via written tests, which can be looked up in the corresponding repositories or the replication kit, and with manually-curated samples of the data. Furthermore, we validated our analysis with a sample (see: Section III-D). Nevertheless, defects in our programs can still exist, which can have an influence on our results.

Furthermore, with our data collection tools, we were not able to check if an imported module is really used in the test. This can be the case, if an import is not removed in a newer version where it is not needed anymore. Nevertheless, we did not see this problem occurring in our sample.

Although, we have tested our regular expressions carefully and with manually-curated samples, they can match wrongly or not at all. The same applies to our created filter. Additionally, we focused on detecting mocks using the official mock library of Python [49]. If other mock libraries are used, we are not able to reliably detect mocks. Nevertheless, none of the projects used other mocking frameworks in our sample.

Furthermore, the modulefinder [51] module of the Python standard library has several shortcomings. First, if the import path is too long, it will throw an exception. Moreover, if there

are syntax errors in the files, the modulefinder module is not able to parse it and therefore it will throw an exception. From the 4,637,232 tests we used in our analysis, an exception was thrown for 202,587, which results in 4,37% of erroneous tests. Note that erroneous tests do not have any imports stored in the database and are therefore excluded from the analysis.

VI. Related Work

In this section, we compare our study to prior work in the field of exploratory unit testing studies.

In the short paper of van Geet et al. [54] the authors determine the adequacy of tests as documentation. They use a dynamic analysis approach to trace the execution of JUnit test cases. Hence, they recorded which method is executed by the test cases. They collected data from three different versions of ant [55]. Their results show, that the number of methods, which are executed by a test case is on average 215. Furthermore, they calculated the method coverage and total number of methods, test methods, and method calls and compared it for each of the three ant versions they analyzed. They concluded, that the developers of ant follow a integration testing, instead of a unit testing strategy. Therefore, following van Geet et al., the tests are "less suitable for documentation purpose" [54].

Our work does have a different focus than the work of van Geet et al. [54]. Whereas van Geet et al. focus on the question, if unit tests can be used as documentation, we focus on the question if there are any unit tests at all in a project. Additionally, we consider the developer's classifications of a test and the usage of mocks and its influence in our analysis. Furthermore, their data basis is small in comparison to ours, as they collected data from three versions of one project, whereas our data basis consists of 10 different projects and their whole revision history. Additionally, van Geet et al. [54] use a dynamic analysis approach in their study, which is different to our static analysis approach. Finally, they focus on projects written in Java, which is in contrast to our work, as we focus on projects written in Python.

Another paper in the field of exploratory unit testing studies is the paper by Gälli et al. [56]. They provide a taxonomy of unit tests with different categories based on one individually classified project, which is tested via a semi-automatic approach that classifies the unit tests of the test project into the different categories of their schema dynamically. It is not clear, if they used only one revision of these projects or the whole history. Moreover, Gälli et al. [56] do not state after which criteria or definition they classify tests as unit tests. Their semi-automatic approach reached a recall of 52% and a precision of 89% for the one category they had a look at.

Their work focuses on projects written in Smalltalk [57], whereas our focus lies on projects written in Python. Furthermore, the focus of the paper by Gälli et al. [56] is different to ours, as they want to provide a taxonomy of unit tests, whereas our focus lies on if there exist any unit tests in the projects.

Runeson [58] investigates the unit testing practices of practitioners in his paper. He surveyed 32 participants of 19 different companies. The survey was organized in two phases: in the first phase the 17 participants held a focus group discussion. In the second phase, the results of these discussions were used as the basis for a questionnaire, which was completed by 15 participants. The analysis of the questionnaire and the discussions were done with the goal to find out what practitioners refer to, when they talk about unit testing. The reporting of the results is structured according to Zachman's framework [59]. Runeson reports for each of the frameworks categories (except "Where?") definitions, strengths, and problems that developers see or use. His conclusion is that companies must define what unit testing means for them, as an unclear definition runs the risk of inconsistent (or bad) testing.

In contrast to our study, Runeson did not analyze the code, which the developers wrote, but surveyed them to get their thoughts about unit testing. Therefore, the focus of both studies is differently, but both studies refer to the same problems.

Besides the mentioned related work, there are a lot of other publications focusing on establishing traceability links between unit tests and units under test, e.g., [6], [7]. Nevertheless, they do not focus on the question, if developers write unit tests in real world projects, but on linking test cases with code that is tested to improve the productivity of developers. To the best of our knowledge, no previous research has empirically studied, if developers write unit tests in real world projects.

VII. Conclusion and Future Work

In this paper, we reported an empirical study conducted on 10 open source projects aimed at investigating how (and whether) developers use unit tests in real world projects. In particular, we investigated: the developers categorization of their own tests, the number of unit tests, if mocks are used and their influence on the results, the evolution of the number of unit tests, and whether the definition of unit testing that is used influences the results.

We mined the complete revision history of the 10 projects and detected and categorized all tests at each of the 70,953 analyzed revisions. This data was further analyzed in respect to our RQs mentioned above. Our results show: (i) developers believe that they are developing unit tests, but in fact they are developing fewer than they think, (ii) in most projects, there is just a small amount of unit tests, but this depends largely on the project, (iii) developers use mocks often, but it does not have a significant influence on the number of unit tests, (iv) four different patterns were detected of how the number of unit tests evolve over time in the examined projects, but for one project no pattern could be observed, (v) the used unit test definition has an influence on the results.

For future work, we are extending our tool so that it can also analyze other programming languages (e.g., Java). Additionally, we plan to have a look at other test types (integration/system tests) to see how they are used in real life projects. Furthermore, we plan to involve the developers of the projects and conduct a survey, as we believe that this could help us to understand what open-source developers really think about unit testing.

References

[1] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[2] ——, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[3] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE software*, vol. 1, no. 1, p. 75, 1984.

[4] A. Spillner, T. Linz, and H. Schaefer, *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, Inc., 2014.

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

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

[7] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, "Ezunit: A framework for associating failed unit tests with potential programming errors," in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2007, pp. 101–104.

[8] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, pp. 287–301, 2001.

[9] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[10] A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, 2001, pp. 92–95.

[11] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[12] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 2007, pp. 213–222.

[13] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *22rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM, 2007, pp. 417–420.

[14] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. IEEE, 2005, pp. 10–pp.

[15] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 193–202.

[16] K. Taneja and T. Xie, "Diffgen: Automated regression unit-test generation," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*. IEEE, 2008, pp. 407–410.

[17] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 364–374.

[18] Pierre Carbonnelle, "Popularity of Programming Languages Index," http://pypl.github.io/PYPL.html, [accessed 01-August-2016].

[19] International Software Testing Qualification Board, "International Software Testing Qualification Board Glossary," http://www.astqb.org/glossary/search/unit, [accessed 01-August-2016].

[20] "Systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, 2010.

[21] Fabian Trautsch, "Replication Kit," https://user.informatik.uni-goettingen.de/~ftrauts/replication_kit_01.tar.gz, [accessed 02-August-2016].

[22] G. K. Thiruvathukal, K. Läufer, and B. Gonzalez, "Unit testing considered useful," *Computing in Science and Engineering*, vol. 8, no. 6, pp. 76–87, 2006.

[23] International Software Testing Qualification Board, "International Software Testing Qualification Board Glossary," http://www.astqb.org/glossary/search/test, [accessed 01-August-2016].

[24] ——, "International Software Testing Qualification Board Glossary," http://www.astqb.org/glossary/search/testcase, [accessed 01-August-2016].

[25] Python Software Foundation, "Python - Import System," https://docs.python.org/3/reference/import.html, [accessed 02-August-2016].

[26] Kenneth Reitz, Tanya Schlusser, "The Hitchhiker's Guide to Python - Structuring," http://docs.python-guide.org/en/latest/writing/structure/, [accessed 02-August-2016].

[27] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[28] F. Q. Da Silva, M. Suassuna, A. C. C. França, A. M. Grubb, T. B. Gouveia, C. V. Monteiro, and I. E. dos Santos, "Replication of empirical studies in software engineering research: a systematic mapping study," *Empirical Software Engineering*, vol. 19, no. 3, pp. 501–557, 2014.

[29] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering," *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, 2008.

[30] MongoDB, Inc., "MongoDB Homepage," https://www.mongodb.com/de, [accessed 02-August-2016].

[31] GitHub, Inc., "Github," https://github.com/, [accessed 02-August-2016].

[32] "awesome-python Github Page," https://github.com/vinta/awesome-python, [accessed 01-August-2016].

[33] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2000, pp. 131–142.

[34] Ansible, Inc., "ansible Github Page," https://github.com/ansible/ansible, [accessed 01-August-2016].

[35] Amazon Web Services, Inc., "aws-cli Github Page," https://github.com/aws/aws-cli, [accessed 01-August-2016].

[36] "nose Github Page," https://github.com/nose-devs/nose, [accessed 01-August-2016].

[37] "nose2 Github Page," https://github.com/nose-devs/nose2, [accessed 01-August-2016].

[38] Numenta, "nupic Github Page," https://github.com/numenta/nupic, [accessed 01-August-2016].

[39] Python Packaging Authority, "pip Github Page," https://github.com/pypa/pip, [accessed 01-August-2016].

[40] "Robot Framework Github Page," https://github.com/robotframework/robotframework, [accessed 01-August-2016].

[41] David Cournapeau, "scikit-learn Github Page," https://github.com/scikit-learn/scikit-learn, [accessed 01-August-2016].

[42] Kevin McCarthy, "vcrpy Github Page," https://github.com/kevin1024/vcrpy, [accessed 01-August-2016].

[43] Python Packaging Authority, "warehouse Github Page," https://github.com/pypa/warehouse, [accessed 01-August-2016].

[44] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski, "Adressing problems with external validity of repository mining studies through a smart data platform," in *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM, 2016, pp. 97–108.

[45] Fabian Trautsch, "vcsSHARK documentation," http://ftrautsch.github.io/vcsSHARK/index.html, [accessed 02-August-2016].

[46] "libgit2 Homepage," https://libgit2.github.com/, [accessed 02-August-2016].

[47] "mongoengine Homepage," http://mongoengine.org/, [accessed 02-August-2016].

[48] Fabian Trautsch, "testImpSHARK Repository," https://gitlab.gwdg.de/ftrauts/testImpSHARK, [accessed 02-August-2016].

[49] Python Software Foundation, "Mock object library," https://docs.python.org/3/library/unittest.mock.html, [accessed 01-August-2016].

[50] Ronald Oussoren, "modulegraph documentation," http://pythonhosted.org/modulegraph/, [accessed 02-August-2016].

[51] Python Software Foundation, "modulefinder documentation," https://docs.python.org/3/library/modulefinder.html, [accessed 02-August-2016].

[52] scikit-learn developers, "Scikit-Learn contribution guide," http://scikit-learn.org/stable/developers/contributing.html#testing-and-improving-test-coverage, [accessed 01-August-2016].

[53] W. G. Cochran, "Sampling techniques." 1977.

[54] J. Van Geet, A. Zaidman, O. Greevy, and A. Hamou-Lhadj, "A lightweight approach to determining the adequacy of tests as documentation," in *Proc. 2nd Int. Workshop on Program Comprehension through Dynamic Analysis (PCODA06)*, 2006, pp. 21–26.

[55] Apache Software Foundation, "Apache Ant Homepage," http://ant.apache.org/, [accessed 02-August-2016].

[56] M. Gaelli, M. Lanza, and O. Nierstrasz, “Towards a taxonomy of unit tests,” in *Proceedings of the International European Smalltalk Conference*. Citeseer, 2005, pp. 1–10.
[57] C. Liu, *Smalltalk, objects, and design*. iUniverse, 2000.
[58] P. Runeson, “A survey of unit testing practices,” *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.
[59] J. A. Zachman, “A framework for information systems architecture,” *IBM systems journal*, vol. 26, no. 3, pp. 276–292, 1987.