Delft University of Technology

To Mock or Not To Mock?
An Empirical Study on Mocking Practices

Spadini, Davide; Aniche, Maurício; Bruntink, Magiel; Bacchelli, Alberto

DOI
10.1109/MSR.2017.61

Publication date
2017

Document Version
Peer reviewed version

Published in
Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017

Citation (APA)
Spadini, D., Aniche, M., Bruntink, M., & Bacchelli, A. (2017). To Mock or Not To Mock? An Empirical Study on Mocking Practices. In Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017 (pp. 402-412). [7962389] Los Alamitos, CA: IEEE Computer Society. DOI: 10.1109/MSR.2017.61

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

To Mock or Not To Mock? An Empirical Study on Mocking Practices

Davide Spadini*†, Maurício Aniche†, Magiel Bruntink*, Alberto Bacchelli†
*Software Improvement Group
{d.spadini, m.bruntink}@sig.eu
†Delft University of Technology
{d.spadini, m.f.aniche, a.bacchelli}@tudelft.nl

***Abstract*—When writing automated unit tests, developers often deal with software artifacts that have several dependencies. In these cases, one has the possibility of either instantiating the dependencies or using mock objects to simulate the dependencies' expected behavior. Even though recent quantitative studies showed that mock objects are widely used in OSS projects, scientific knowledge is still lacking on how and why practitioners use mocks. Such a knowledge is fundamental to guide further research on this widespread practice and inform the design of tools and processes to improve it.**

The objective of this paper is to increase our understanding of which test dependencies developers (do not) mock and why, as well as what challenges developers face with this practice. To this aim, we create MOCKEXTRACTOR, a tool to mine the usage of mock objects in testing code and employ it to collect data from three OSS projects and one industrial system. Sampling from this data, we manually analyze how more than 2,000 test dependencies are treated. Subsequently, we discuss our findings with developers from these systems, identifying practices, rationales, and challenges. These results are supported by a structured survey with more than 100 professionals. The study reveals that the usage of mocks is highly dependent on the responsibility and the architectural concern of the class. Developers report to frequently mock dependencies that make testing difficult and prefer to not mock classes that encapsulate domain concepts/rules of the system. Among the key challenges, developers report that maintaining the behavior of the mock compatible with the behavior of original class is hard and that mocking increases the coupling between the test and the production code.

I. INTRODUCTION

In software testing, it is common that the software artifact under test depends on other units [36]. Therefore, when testing a unit (*e.g.*, a class in object-oriented programming), developers often need to decide whether to test the unit and all its dependencies together (similar to integration testing) or to simulate these dependencies and test that unit in isolation.

By testing all dependencies together, developers gain realism: The test will more likely reflect the behavior in production [41]. However, some dependencies, such as databases and web services, may (1) slow the execution of the test [31], (2) be costly to properly setup for testing [37], and (3) require testers to have full control over such external dependencies [18]. By simulating its dependencies, developers gain focus: The test will cover only the specific unit and the expected interactions with its dependencies; moreover, inefficiencies of testing dependencies are mitigated.

To support the simulation of dependencies, *mocking frameworks* have been developed (*e.g.*, Mockito [7], EasyMock [2], and JMock [3] for Java, Mock [5] and Mocker [6] for Python), which provide APIs for creating mock (*i.e.*, simulated) objects, setting return values of methods in the mock objects, and checking interactions between the component under test and the mock objects. Past research has reported that software projects are using mocking frameworks widely [21] [32] and has provided initial evidence that using a mock object can ease the process of unit testing [29].

However, empirical knowledge is still lacking on *how* and *why* practitioners use mocks. To scientifically evaluate mocking and its effects, as well as to help practitioners in their software testing phase, one has to first understand and quantify developers' practices and perspectives. In fact, this allows both to focus future research on the most relevant aspects of mocking and on real developers' needs, as well as to effectively guide the design of tools and processes.

To fill this gap of knowledge, the goal of this paper is to *empirically understand how and why developers apply mock objects* in their test suites. To this aim, we analyzed more than 2,000 test dependencies from three OSS projects and one industrial system. We then interviewed developers from these systems to understand why some dependencies were mocked and others were not. We challenged and supported our findings by surveying 105 developers from software testing communities. Finally, we discussed our findings with a main developer from the most used Java mocking framework.

The *main contributions* of this paper are:

1) A categorization of the most often mocked and not mocked dependencies, based on a quantitative analysis on three OSS systems and one industrial system (RQ_1).
2) An empirical understanding of why and when developers mock, after interviewing developers of analyzed systems and surveying 105 developers (RQ_2).
3) The main challenges faced by developers when making use of mock objects in the test suites, also extracted from the interviews and surveys (RQ_3).
4) An open source tool, namely MOCKEXTRACTOR, that is able to extract the set of mocked and non mocked dependencies in a given Java test suite. The tool is available in our on-line appendix [12] and GitHub.

II. Background

"Once," said the Mock Turtle at last, with a deep sigh, "I was a real Turtle."
— Alice In Wonderland, Lewis Carroll

A. Mock objects

Mock objects are used to replace real software dependencies by simulating their relevant features [28]. Typically, methods of mock objects are designed to return some desired values given specific input values. Listing II-A shows an example usage of Mockito, one of the most popular mocking libraries in Java [32]. We now explain each code block of the example:

1) At the beginning, one must define the class that should be mocked by Mockito. In our example, *LinkedList* is being mocked (line 2). The returned object (*mockedList*) is now a mock: It can respond to all existing methods in the *LinkedList* class.
2) As second step, we provide a new behaviour to the newly instantiated mock. In the example, we inform the mock to return the string 'first' when *mockedList.get(0)* is invoked (line 5) and to throw a *RuntimeException* on *mockedList.get(1)* (line 7).
3) The mock is now ready to be used. In line 10 and 11 the mock will answer method invocations with the values provided in step 2.

```
1  // 1: Mocking LinkedList
2  LinkedList mockObj = mock(LinkedList.class);
3
4  // 2: Instructing the mock object behaviour
5  when(mockObj.get(0)).thenReturn("first");
6  when(mockObj.get(1))
7    .thenThrow(new RuntimeException());
8
9  // 3: Invoking methods in the mock
10 System.out.println(mockObj.get(0));
11 System.out.println(mockObj.get(1));
```

Listing 1: Example of an object being mocked

Overall, whenever developers do not want to rely on the real implementation of a dependency, (*e.g.*, to isolate a unit test) they can simulate it and define the expected behavior using the aforementioned approach.

B. Motivating example

Sonarqube is a popular open source system that provides continuous code inspection [10]. In January of 2017, Sonarqube contained over 5,500 classes, 700k lines of code, and 2,034 test units. Among all test units, 652 make use of mock objects, mocking a total of 1,411 unique dependencies.

Let us consider the class *IssueChangeDao* as an example. This class is responsible for accessing the database regarding changes in issues (changes and issues are business entities of the system). To that end, this class uses MyBatis [8], a Java library for accessing databases.

There are four test units that use *IssueChangeDao*. The dependency is mocked in two of them; in the other two, the test creates a concrete instance of the database (to access the database during the test execution). *Why do developers mock the dependency in somes cases and do not mock in other cases?* Indeed, this is a key question motivating this work.

After manually analyzing these tests, we observed that:

- In Test 1, the class is concretely instantiated as this test unit performs an integration test with one of their web services. As the test exercises the web service, a database needs to be active.
- In Test 2, the class is also concretely instantiated as *IssueChangeDao* is the class under test.
- In both Test 3 and Test 4, test units focus on testing two different classes that use *IssueChangeDao* as part of their job.

This single example shows us that developers may have different reasons to mock or not mock a class. In the remainder of this paper, we investigate patterns of how developers mock by analyzing the use of mocks in software systems and we investigate their rationale by interviewing and surveying practitioners on their mocking practices.

III. Reseach Methodology

The *goal* of our study is to understand how and why developers apply mock objects in their test suites. To that end, we conduct quantitative and qualitative research focusing on four software systems and address the following questions:

RQ_1: What test dependencies do developers mock? When writing an automated test for a given class, developers can either mock or use a concrete instance of its dependencies. Different authors [28], [17] affirm that mock objects can be used when a class depends upon some infrastructure (*e.g.*, file system, caching). We aim to identify what dependencies developers mock by means of manual analysis in source code from different systems.

RQ_2: Why do developers decide to (not) mock specific dependencies? We aim to find an explanation to the findings in previous RQ. We interview developers from the analyzed systems and ask for an explanation on why some dependencies are mocked while others are not. Furthermore, we survey software developers with the goal of challenging the findings from the interviews.

RQ_3: Which are the main challenges experienced with testing using mocks? Understanding challenges sheds a light on important aspects on which researchers and practitioners can effectively focus next. Therefore, we investigate the main challenges developers face using mocks by means of interviews and surveys.

A. Sample selection

We focus on projects that routinely use mock objects. We analyze projects that make use of Mockito, the most popular framework in Java with OSS projects [32].

We select three open source software projects (*i.e.*, Sonarqube [10], Spring [11], VRaptor [13]) and a software system from an industrial organization we previously collaborated with (Alura [1]); Table I details their size. In the following, we describe their suitability to our investigation.

TABLE I: Description of our studied sample (N=4).

Project	# of classes	LOC	# of test units	# of test units with mock	# of mocked dependencies	# of not mocked dependencies	Sample size of mocked (CL=95%)	Sample size of not mocked (CL=95%)
Sonarqube	5,771	701k	2,034	652	1,411	12,136	302	372
Spring Framework	6561	997k	2,020	299	670	21,098	244	377
VRaptor	551	45k	126	80	258	1,075	155	283
Alura	1,009	75k	239	91	229	1,436	143	302
Total	13.892	1.818k	4.419	1.122	2,568	35,745	844	1,334

Spring Framework. Spring provides an extensive infrastructural support for Java developers; its core serves as a base for many other offered services, such as dependency injection and transaction management.

Sonarqube. Sonarqube is a quality management platform that continuously measures the quality of source code and delivers reports to its developers.

VRaptor. VRaptor is an MVC framework that provides an easy way to integrate Java EE capabilities (such as CDI) and to develop REST webservices.

Alura. Alura is a proprietary web e-learning system used by thousands of students and teachers in Brazil; it is a database-centric system developed in Java.

B. Data Collection and Analysis

The research method we used to answer our research questions follows a mixed qualitative and quantitative approach, which we depict in Figure 1: (**1**) We automatically collected all mocked and non-mocked dependencies in the test units of the analyzed systems, (**2**) we manually analyzed a sample of these dependencies with the goal of understanding their architectural concerns as well as their implementation, (**3**) we grouped these architectural concerns into categories, which enabled us to compare mocked and non mocked dependencies among these categories, (**4**) we interviewed developers from the studied systems to understand our findings, and (**5**) we enhanced our results in a on-line survey with 105 respondents.

1. Data collection. To obtain data on mocking practices, we first collected all the dependencies in the test units of our systems performing static analysis on their test code. To this aim, we created the tool MOCKEXTRACTOR [38], which implements the algorithm below:

1) We detect all test classes in the software system. As done in past literature (*e.g.*, Zaidman *et al.* [43]), we consider a class to be a test when its name ends with 'Test' or 'Tests'.
2) For each test class, we extract the (possibly extensive) list of all its dependencies. Examples of dependencies are the class under test itself, its required dependencies, and utility classes (*e.g.*, lists and test helpers).
3) We mark each dependency as 'mocked' or 'not mocked'. Mockito provides two APIs for creating a mock from a given class:[1] (1) By making use of the `@Mock` annotation in a class field or (2) by invoking `Mockito.mock()` inside the test method. Every time one of the two options is found in the code, we identify the type of the class that is mocked. The class is then marked as 'mocked' in that test unit. If a dependency appears more than once in the test unit, we consider it 'mocked'. A dependency may be considered 'mocked' in one test unit, but 'not mocked' in another.
4) We mark dependencies as 'not mocked' by subtracting the mocked dependencies from the set of all dependencies.

2. Manual analysis. To answer what test dependencies developers mock, we analyzed the previously extracted mocked and non mocked dependencies. The goal of the analysis is to understand the main concern of the class in the architecture of the software system (*e.g.*, a class is responsible for representing a business entity, or a class is responsible for persisting into the database). Defining the architectural concern of a class is not an easy task to be automated, since it is context-specific [12], thus we decided to perform a manual analysis. The first two authors of the paper conducted this analysis after having studied the architecture of the four systems.

Due to the size of the total number of mocked and non mocked dependencies (~38,000), we analyzed a random sample. The sample is created with the confidence level of 95% and the error (E) of 5%, *i.e.*, if in the sample a specific dependency is mocked $f\%$ of the times, we are 95% confident that it will be mocked $f\% \pm 5\%$ in the entire test suite. Since projects belong to different areas and results can be completely different from each other, we created a sample for each project. We produced four samples, one belonging to each project. This gave us fine-grained information to investigate mock practices within each project.

In Table I we show the final number of analyzed dependencies ($844 + 1,334 = 2,178$ dependencies).

The manual analysis procedure was as follows:

- Each researcher was in charge of two projects. The selection was done by convenience: The second author was already familiar with the internal structure of VRaptor and Alura.
- All dependencies in the sample were listed in a spreadsheet in which both researchers had access. Each row contained information about the test unit that dependency was found and a boolean indicating if that dependency was mocked.
- For each dependency in the sample, the researcher manually inspected the source code of the class. To fully understand the class' architectural concern, researchers were allowed to navigate through any other relevant piece of code.
- After understanding the concern of that class, the researcher filled the "Category" column with what best describes the

[1]Mockito can also generate *spies* which are out of the scope of this paper. More information can be found at Mockito's documentation: http://bit.ly/2kjtEi6.

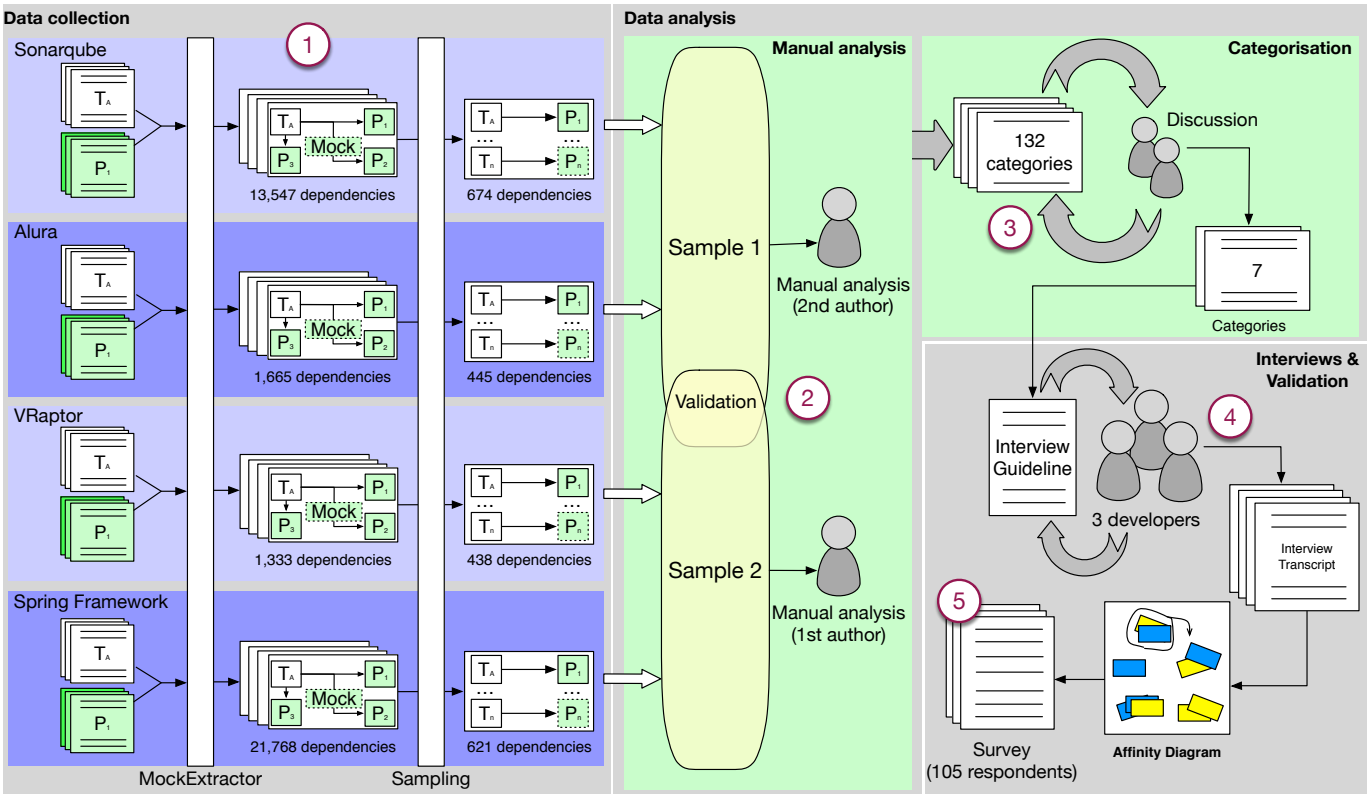

Fig. 1: The mixed approach research method applied.

concern. No categories were defined up-front. In case of doubt, the researcher first read the test unit code; if not enough, he then talked with the other research.

- At the end of each day, the researchers discussed together their main findings and some specific cases.

The full process took seven full days. The total number of categories was 116. We then started the second phase of the manual analysis, focused on *merging categories*.

3. Categorization. To group similar categories we used a technique similar to card sort [35]: (**1**) each category represented a card, (**2**) the first two authors analyzed the cards applying open (*i.e.*, without predefined groups) card sort, (**3**) the researcher who created the category explained the reasons behind it and discussed a possible generalization (making the discussion more concrete by showing the source code of the class was allowed during the discussion), (**4**) similar categories were then grouped into a final, higher level category. (**5**) at the end the authors gave a name to each *final* category.

After following this procedure for all the 116 categories, we obtained a total of 7 categories that describe the concerns of classes.

The large difference between 116 and 7 is the result of most concerns being grouped into two categories: 'Domain object' and 'External dependencies'. The former classes always represented some business logic of the system and had no external dependencies. The full list of the 116 categories is available in our on-line appendix [12].

TABLE II: Profile of the interviewees

Project	ID	Role in the project	Years of programming experience
Spring Framework	D1	Lead Developer	25
VRaptor	D2	Developer	10
Alura	D3	Lead Developer	5

4. Interviews. We used results from the previous RQ as an input to the data collection procedure of RQ_2. We designed an interview in which the goal was to understand *why* developers did mock some roles and did not mock other roles. The interview was semi-structured and was conducted by the first two authors of this paper. For each finding in previous RQ, we made sure that the interviewee described why they do (or do not) mock that particular category, what the perceived advantages and disadvantages are, and any exceptions to this rule. Our full interview protocol is available in the appendix [12].

We conducted 3 interviews with active, prolific developers from 3 projects (unfortunately no developer from Sonarqube was available for an interview). Table II shows the interviewees' details.

We started each interview by asking general questions about mocking practices. More specifically, we were interested in understanding why and what classes they commonly mock. Afterwards, we focused on the results gathered by answering the previous RQ. We presented the interviewee with two tables: one containing the results of all projects (Figure 2) and another one containing only the results of the interviewee's

project. For each category, we presented the findings and solicit an interpretation (*e.g.*, by explaining why it happens in their specific project and by comparing with what we saw in other projects). From a high-level perspective, we asked:

1) Can you explain this difference? Please, think about your experience with this project in particular.
2) We observe that your numbers are different when compared to other projects. In your opinion, why does it happen?
3) In your experience, when should one mock a <category>? Why?
4) In your experience, when should one not mock a <category>? Why?
5) Are there exceptions?
6) Do you know if your rules are also followed by the other developers in your project?

Throughout the interview, one of the researchers was in charge of summarizing the answers. Before finalizing the interview, we revisited the answers with the interviewee to validate our interpretation of their opinions. Finally, we asked questions about challenges with mocking.

Interviews were conducted via Skype and fully recorded. Each of them was manually transcribed by the researchers. With the full transcriptions, we performed card sorting [39], [20] to identify the main themes.

As a complement to the research question, whenever feasible, we also validated interviewees' perceptions by measuring them in their own software system.

5. Survey. To challenge and expand the concepts that emerged during the previous phases, we conducted a survey. All questions were derived from the results of previous RQs. The survey had four main parts. In the first part, we asked participants about their experience in software development and mocking. The second part of the survey asked participants about how often they make use of mock objects in each of the categories found during the manual analysis. The third part asked participants about how often they mock classes in specific situations, such as when the class is too complex or coupled. The fourth part was focused on asking participants about challenges with mocking. Except for this last question, which was open-ended and optional, all the other questions were closed-ended and participants had to choose between a 5-point Likert scale.

The survey was initially designed in English. We compiled Brazilian Portuguese translation, to reach a broader, more diverse population. Before deploying the survey, we first performed a pilot of both versions with four participants; we improved our survey based on their feedbacks (changes were all related to phrasing). We then shared our survey via Twitter (authors tweeted in their respective accounts), among our contacts, and in developers' mailing lists. The survey ran for one week. We analyzed the open questions by also performing card sorting. The full survey can be found in our on-line appendix [12].

We received a total of 105 answers from both Brazilian Portuguese and English surveys. 21% of the respondents have between 1 and 5 years of experience, 64% between 6 and 15 and 15% have more than 15 years of experience. The most used programming language is Java (24%), the second is JavaScript (19%) and the third one is C# (18%). The mocking framework most used by the respondents is Mockito (33%) followed by Moq (19%) and Powermock (5%).

C. Threats to Validity

Our methodology may pose some threats to the validity of the results we report in Section IV. We discuss them here.

1) Construct validity: Threats to *construct validity* concern our research instruments. We develop and use MOCKEXTRACTOR to collect dependencies that are mocked in a test unit by means of static code analysis. As with any static code analysis tool, MOCKEXTRACTOR is not able to capture dynamic behavior (*e.g.*, mock instances that are generated in helper classes and passed to the test unit). In these cases, the dependency would have been considered "non mocked". We mitigate this issue by (1) making use of a large random samples in our manual analysis, and (2) manually inspecting the results of MOCKEXTRACTOR in 100 test units, in which we observed that such cases never occurred, thus giving us confidence regarding the reliability of our data set.

As each class is manually analyzed by only a single researcher and there could be divergent opinions despite the aforementioned discussion, we measured their agreement. Each researcher analyzed 25 instances that were made by the other researcher in both of his two projects, totaling 100 validated instances as seen in Figure 1, Point 2. The final agreement on the 7 categories was 89%.

2) Internal validity: Threats to *internal validity* concern factors we did not consider that could affect the variables and the relations being investigated. In our study, we interview developers from the studied software to understand why certain dependencies are mocked and not mocked. Clearly, a single developer does not know all the implementation decisions in a software system. We mitigate this issue by (1) showing the data collected in RQ1 first and (2) not asking questions about the overall categories that we manually coined in RQ1.

In addition, their opinions may also be influenced by other factors, such as current literature on mocking (which could may have led them to social desirability bias [33]) or other projects that they participate in. To mitigate this issue, we constantly reminded interviewees that we were discussing the mocking practices specifically of their project. At the end of the interview, we asked them to freely talk about their ideas on mocking in general.

3) External validity: Threats to *external validity* concern the generalization of results. Our sample contains four Java systems (one of them closed source), which is small compared to the overall population of software systems that make use of mocking. We reduce this issue by collecting the opinion of 105 developers from a variety of projects about our findings. Further research in different projects in different programming languages should be conducted.

Furthermore, we do not know the nature of the population that responded to our survey, hence it might suffer from a self-

selection bias. We cannot calculate the response rate of our survey; however, from the responses we see a general diversity in terms of software development experience that appears to match in our target population.

IV. Results

In this section, we present the results to our research questions aimed at understanding how and why developers apply mock objects in their test suites, as well as which challenges they face in this context.

RQ1. What test dependencies do developers mock?

As we show in Table I, we analyzed 4,419 test units of which 1,122 (25.39%) contain at least one mock object. From the 38,313 collected dependencies from all test units, 35,745 (93.29%) are not mocked while 2,568 (6.71%) are mocked.

As the same dependency may appear more than once in our dataset (*i.e.*, , a class can appear in multiple test units), we calculated the *unique* dependencies in our dataset. We obtained a total of 11,824 not mocked and 938 mocked dependencies. Interestingly, the intersection of these two sets reveals that 650 dependencies (70% of all dependencies mocked at least once) were both mocked and not mocked in the test suite.

In Figure 2, we show how often each role is mocked in our sample in each of the seven categories found during our manual analysis. One may note that "databases" and "web services" can also fit in the "external dependency" category; we separate these two categories as they appeared more frequently than other types of external dependencies.

In the following, we explain each category:

- **Domain object:** Classes that contain the (business) rules of the system. Most of these classes usually depend on other domain objects. They do not depend on any external resources. The definition of this category fits well to the definition of Domain Object [15] and Domain Logic [16] architectural layers. Examples are entities, services and utility classes.
- **Database:** Classes that interact with an external database. These classes can be either an external library (such as Java SQL, JDBC, Hibernate, or ElasticSearch APIs) or a class that depends on such external libraries (*e.g.*, an implementation of the Data Access Object [16] pattern).
- **Native Java libraries:** Libraries that are part of the Java itself. Examples are classes from Java I/O and Java Util classes (Date, Calendar).
- **Web Service:** Classes that perform some HTTP action. As with the database category, this dependency can be either an external library (such as Java HTTP) or a class that depends on such library.
- **External dependency:** Libraries (or classes that make use of libraries) that are external to the current project. Examples are Jetty and Ruby runtimes, JSON parsing libraries (such as GSON), e-mail libraries, etc.
- **Test support:** Classes that support testing itself. Examples are fake domain objects, test data builders and web services for tests.

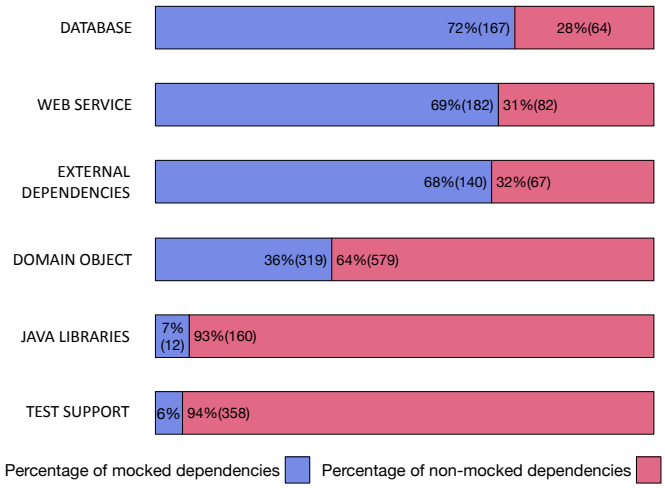

Fig. 2: How often each architectural role is mocked and not mocked in analyzed systems ($N = 2,178$)

- **Unresolved:** Dependencies that we were not able to solve. For example, classes belonging to a sub-module of the project which the source code is not available.

Numbers are quite similar when we look at each project separately. Exceptions are for databases (Alura and Sonarqube mock ~60% of databases dependencies, Spring mocks 94%) and domain objects (while other projects mock them ~30% of times, Sonarqube mocks 47%). We present the numbers for each project in our online appendix [12].

We observe that *Web Services* and *Databases* are the most mocked dependencies. On the other hand, there is no clear trend in *Domain objects*: numbers show that 36% of them are mocked. Even though the findings are aligned with the technical literature [28], [23], further investigation is necessary to understand the real rationale behind the results.

In contrast *Test support* and *Java libraries* are almost never mocked. The former is unsurprising since the category includes fake classes or classes that are created to support the test itself.

> *RQ_1. Classes that deal with external resources, such as databases and web services are often mocked. Interestingly, there is no clear trend on mocking domain objects.*

RQ2. Why do developers decide to (not) mock specific dependencies?

In this section, we summarize the answers obtained during our interviews and surveys. We refer to the interviewees by their ID in Table II.

Mocks are often used when the concrete implementation is not simple. All interviewees agree that certain dependencies are easier to mock than to use their concrete implementation. They mentioned that classes that are highly coupled, complex to set up, contain complex code, perform a slow task, or depend on external resources (e.g., databases, web services or external libraries) are candidates to be mocked. D2 gives a

concrete example: *"It is simpler to set up a in-memory list with elements than inserting data into the database."* Interviewees affirmed that whenever they can completely control the input and output of a class, they prefer to instantiate the concrete implementation of the class rather than mocking it. As D1 stated: *"if given an input* [the production class] *will always return a single output, we do not mock it."*

In Figure 3, we see that survey respondents also often mock dependencies with such characteristics: 48% of respondents said they always or almost always mock classes that are highly coupled, and 45.5% when the class difficult to set up. Contrarily to our interviewees, survey respondents report to mock less often when it comes to slow or complex classes (50.4% and 34.5% of respondents affirm to never or almost never mock in such situations, respectively).

Mocks are not used when the focus of the test is the integration. Interviewees explained that they do not use mocks when they want to test the integration with an external dependency itself, (*e.g.*, a class that integrates with a database). In these cases they prefer to perform a real interaction between the unit under test and the external dependency. D1 said *"if we mock* [the integration]*, then we wouldn't know if it actually works.* [...] *I do not mock when I want to test the database itself; I wanna make sure that my SQL works. Other than that, we mock."* This is also confirmed in our survey (Figure 3), as our respondents also almost never mock the class under test.

The opposite scenario is when developers want to *unit* test a class that depends on a class that deals with external resources, (*e.g.*, *Foo* depends on *Boo*, and *Boo* interacts with a database). In this case, developers want to test a single unit without the influence of the external dependencies, thus developers evaluate whether they should mock that dependency. D2 said: *"in unit testing, when the unit I wanna test uses classes that integrate with the external environment, we do not want to test if the integration works, but if our current unit works,* [...] *so we mock the dependencies."*

Interfaces are mocked rather than one of their specific implementations. Interviewees agree that they often mock interfaces. They explain that an interface can have several implementations and they prefer to use a mock to not rely on a specific one. D1 said: *"when I test operations with side effects* [sending an email, doing a HTTP Request] *I create an interface that represents the side effect and* [instead of using a specific implementation] *I mock directly the interface."*

Domain objects are usually not mocked. According to the interviewees, domain objects are often plain old Java objects, commonly composed by a set of attributes, getters and setters. These classes also commonly do not deal with external resources. Thus, these classes tend to be easily instantiated and set up. However, if a domain object is complex (*i.e.*, contains complicated business logic or not easy to set up), developers may mock them. Interviewee D2 says: "[if class A depends on the domain object B] *I'd probably have a BTest testing B so this is a green light for me to know that I don't need to test B again."* All interviewees also mention that the same rule applies if the domain object is highly coupled.

Figure 4 shows that answers about mocking *Domain objects* vary. Interestingly, there is a slight trend towards not mocking them, in line to our findings during the interviews and in RQ1.

Native Java objects and libraries are usually not mocked. According to D1, native Java objects are data holders (*e.g.*, String and List) that are easy to instantiate with the desired value. Thus no need for mocking. D1 points out that some native classes cannot even be mocked as they can be final (*e.g.*, String). D2 discussed the question from a different perspective. According to him, developers can trust the provided libraries, even though they are "external," thus, there is no need for mocking. Both D1 and D2 made an exception for the Java I/O library: According to them, dealing with files can also be complex, and thus, they prefer to mock. D3, on the other hand, affirms that in their software, they commonly do not mock I/O as they favor integration testing.

These findings match our data from RQ1, where we see that *Native Java Libraries* are almost never mocked. Respondents also had a similar perception: 82% of them affirm to never or almost never mock such dependencies.

Database, web services, and external dependencies are slow, complex to set up, and are good candidates to be mocked. According to the interviewees, that is why mocks should be applied in such dependencies. D2 said: *"Our database integration tests take 40 minutes to execute, it is too much"*. These reasons also matches with technical literature [28], [23].

All participants have a similar opinion when it comes to other kinds of external dependencies/libraries, such as CDI or a serialization library: When the focus of the testing is the integration itself, they do not mock. Otherwise, they mock. D2 said: *"When using CDI [Java's Contexts and Dependency Injection API], it is really hard to create a concrete [CDI] event: in this case we usually prefer to mock it"*. Two interviewees (D1 and D2) affirmed that libraries commonly have extensive test suites, thus developers do not need to "re-test". D3 had a different opinion: Developers should re-test the library as they cannot always be trusted.

In Figure 4, we observe that respondents always or almost always mock *Web services* (~82%), *External dependencies* (~79%) and *Databases* (~71%). This result confirm the previous discovery that when developers do not want to test the integration itself, they prefer to mock these dependencies.

> ***RQ$_2$.*** *The architectural role of the class is not the only factor developers take into account when mocking. Respondents report to mock when to use the concrete implementation would be not simple, e.g., the class would be too slow or complex to set up.*

RQ3. Which are the main challenges experienced with testing using mocks?

We summarize the main challenges that appeared in the interviews and in the answers of our question about challenges in the survey (which we received 61 answers). Categories

Fig. 3: Reasons to use mock objects ($N = 105$)

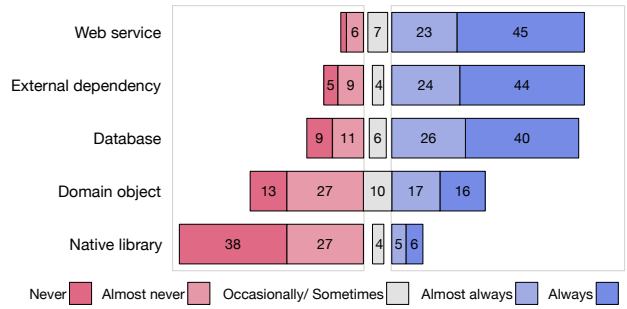

Fig. 4: Frequency of mocking objects per category ($N = 105$)

below represent the main themes that emerged during card sorting.

Dealing with coupling. Mocking practices deal with different coupling issues. On one hand, the usage of mocks in test increases the coupling between the test and the production code. On the other hand, the coupling among production classes themselves can also be challenging for mocking. According to a participant, *"if code has not been written with proper decoupling and dependency isolation, then mocking is difficult (if not impossible)."* This matches with another participant's opinions who mentions to not have challenges anymore, by having *"learned how to separate concepts."*

Getting started with mocks. Mocks can still be a new concept for many developers. Hence, its usage may require experienced developers to teach junior developers (which, according to another participant, usually tend to mock too much). In particular, a participant said that mock objects are currently a new concept for him/her, and thus, s/he is having some trouble understanding it.

Mocking in legacy systems. Legacy systems can pose some challenges for users of mocks. According to a respondent, testing a single unit in such systems may require too much mocking (*"to mock almost the entire system"*). Another participant even mentions the need of using *PowerMock* [9] (a framework that enables Java developers to mock certain classes that might be not possible without bytecode manipulation, *e.g.*, final classes and static methods) in cases where the class under test is not designed for testability. On the other hand, mocking may be the only way to perform unit testing in such systems. According to a participant: *"in legacy systems, where the architecture is not well-decoupled, mocking is the only way to perform some testing."*

Non-testable/Hard-to-test classes. Some technical details may impede the usage of mock objects. Besides the lack of design by testability, participants provide different examples of implementation details that can interfere with mocking. Respondents mentioned the use of static methods in Java (which are not mockable by default), file uploads in PHP, interfaces in dynamic languages, and the LINQ language feature in C#.

The relationship between mocks and good quality code. Mocks may reduce test readability and be difficult to maintain. Survey respondents state that the excessive use of mocks is an indicative of poorly engineered code. Surprisingly during the interviews, D1, D2 and D3 mentioned *the same example* where using mocks can hide a deeper problem in the system's design: *"when you have to test class A, and you notice that it has 10/15 dependencies, you can mock it. However, you are hiding a problem: a class with 15 dependencies is probably a smell in the code."* In this scenario they find it much easier to mock the dependency as it is highly coupled and complex. However, they say this is a symptom of a badly designed class. D3 added: *"good* [production] *code ease the process of testing. If the* [production] *code structure is well defined, we should use less mocks"*. Interviewee D3 also said *"I always try to use as less mocks as possible, since in my opinion they hide the real problem. Furthermore, I do not remember a single case in which I found a bug using mocks'*. A survey respondent also shares the point that the use of mocks does not guarantee that your code will behave as expected in production: *"You are always guessing that what you mock will work (and keep working) that way when using the real objects."*

Unstable dependencies. A problem when using mocks is maintaining the behavior of the mock compatible with the behavior of original class, especially when the class is poorly designed or highly coupled. As the production class tends to change often, the mock object becomes unstable and, as a consequence, more prone to change.

> ***RQ$_3$.*** *The use of mocks poses several challenges. Among all, a major problem is maintaining the behavior of the mock compatible with the original class. Furthermore, mocks may hide important design problems. Finally, while mocking may be the only way to test legacy systems, using them in such systems is not a straightforward task.*

V. Discussion

In this section we discuss the main findings and their implications for both practitioners and future research. We also present the results of a debate about our findings with a main developer from Mockito. Finally, we provide an initial discussion on quantitatively mining mocking practices.

A. Empirical evidence on mocking practices

Mocking is a popular topic among software developers. Due to its importance, different authors have been writing technical literature on mock objects (*e.g.*, [19], [31], [18], [34], [24],

[27]), ranging from how to get started with mocks to best practices. Our research complements such technical literature in three ways that we discuss below.

First, we provide concrete evidence on which of the existing practices in technical literature developers actually apply. For example, Meszaros [31] suggests that components that make testing difficult are candidates to be mocked. Our research confirms it by showing that developers also believe these dependencies should be mocked (RQ2) and that, in practice, developers do mock them (RQ1).

Second, by providing a deeper investigation on how and why developers use mock objects. As a side effect, we also notice how the use of mock objects can drive the developer's testing strategy. For instance, mocking an interface rather than using one concrete implementation makes the test to become "independent of a specific implementation", as the test exercises the abstract behavior that is offered by the interface. Without the usage of a mock, developers would have to choose one out of the many possible implementations of the interface, making the test more coupled to the specific implementation. The use of mock objects can also drive developers towards a better design: Our findings show that a class that requires too much mocking could have been better designed to avoid that. Interestingly, the idea of using the feedback of the test code to improve the quality of production code is popular among TDD practitioners [14].

Third, by providing a list of challenges that can be tackled by researchers, practitioners, and tool makers. Most challenges faced by developers are purely technical, such as applying mocks in legacy systems and in poorly-designed classes, or even dealing with unstable production classes. Interestingly, none of the participants complained about the framework itself (*e.g.*, missing features or bugs).

B. Discussing with a developer from Mockito

To get an even deeper understanding of our results and challenge our conclusions, we interviewed a developer from Mockito, showing him the findings and discussing the challenges. We refer to him as D4.

D4 agreed on the findings regarding what developers should mock: According to him, databases and external dependencies should be mocked when developers do not test the integration itself, while Java libraries and data holders classes should never be mocked instead. Furthermore, D4 also approved what we discovered regarding mocking practices. He affirmed that a good practice is to mock interfaces instead of real classes and that developers should not mock the unit under test. When we argued whether Mockito could provide a feature to ease the mocking process of any of the analyzed categories (Figure 2), he stated: *"If someone tells us that s/he is spending 100 boilerplate lines of code to mock a dependency, we can provide a better way to do it.* [...] *But for now, I can not see how to provide specific features for databases and web services, as Mockito only sees the interface of the class, and not its internal behavior."*

After, we focused on the challenges, as we conjecture that it is the most important and useful part for practitioners and future research and that his experience can shed a light on them. D4 agreed with all the challenges specified by our respondents. When discussing how Mockito could help developers with all the coupling challenges (unstable dependencies, highly coupled classes), he affirmed that the tool itself can not help and that the issue should be fixed in the production class: *"When a developer has to mock a lot of dependencies just to test a single unit, he can do it! However, it is a big red flag that the unit under test is not well designed."*. This reinforces the relationship between the excessive use of mocks and code quality.

When we discussed with him about a possible support for legacy systems in Mockito, D4 said Mockito developers have a philosophical debate internally: They want to keep a clear line of what this framework should and should not do. Non supported features such as the possibility of mocking a static method would enable developers to test their legacy code more easily. However, he stated: *"I think the problem is not adding this feature to Mockito, probably it will require just a week of work, the problem is: should we really do it? If we do it, we allow developers to write bad code."* He also said that final classes can be mocked in Mockito 2.0; interestingly, the feature was not motivated by a willingness to ease the testing of legacy systems, but by developers using Kotlin language [4], in which every class is final by default.

To face the challenge of getting started with mocks, D4 mentioned that Mockito documentation is already extensive and provides several examples on how to better use the framework. However, according to him, knowing what should be mocked and what should not be mocked comes with experience.

C. Quantitatively mining mocking practices

Our study sheds lights on some of the most used practices of mocking objects for testing and their reasons. Work can be done to check and generalize some of the answers given by developers by means of software data mining. This would have the advantage of a more objective view and quick generalizability to other systems. We take a first step into this direction by conducting an initial analysis to test the water and see whether some of our qualitative findings can be confirmed/denied by means of software data mining. In the following paragraphs, we discuss the results of our initial analysis and we provide possible alternatives to mine this information from code repositories.

The unit under test is never mocked. To confirm this assertion, we automatically analyzed all test classes. For each test unit, we verified whether the unit under test (e.g. class *A* in the test unit *ATest*) has been mocked or not. Results show that over ~38,000 analyzed dependencies the unit under test is never mocked in any of the projects.

Unless it is the unit under test, database dependencies are always mocked. To confirm this assumption, for each database dependency (information retrieved from our previous manual

analysis in RQ_1) outside its own test, we counted the number of times in which the dependency was not mocked. In case of Alura, we found that 90% of database dependencies are mocked when not in their specific test unit. When extending this result to all the projects, we obtain an average of 81%.

Complex and coupled classes should be mocked. We take into account two metrics: CBO (Coupling between objects) and McCabe's complexity [30]. We choose these metrics since they have been widely discussed during the interviews. Furthermore, as pointed out during the surveys, developers mock when classes are very coupled or difficult to set up.

With the metrics value for each production class in the four systems, we compare the values from classes that are mocked with the values from classes that are not mocked. In general, as a class can be mocked and not mocked multiple times, we apply a simple heuristic to decide in which category it should belong: If the class has been mocked more than 50% of the times, we put it in the 'mocked' category, and vice-versa (*e.g.*, if a class has been mocked 5 times and not mocked 3 times, it will be categorized as 'mocked'). To compare the two sets, we use the Wilcoxon rank sum test [42] (with confidence level of 95%) and Cliff's delta [22] to measure the effect size. We choose Wilcoxon since it is a non-parametric test (does not have any assumption on the underlying data distribution).

As a result, we see that both mocked and non mocked classes are similar in terms of coupling: The mean coupling of mocked classes is 5.89 with a maximum of 51, while the mean coupling of non mocked classes is even slightly higher (7.131) with a maximum of 187. However, from the Wilcoxon rank sum test and the effect size, we observe that the overall difference is negligible (Wilcoxon p-value<0.001, Cliff's Delta$=-0.121$). Same happens for the complexity metrics: The mean complexity of mocked classes is 10.58 with a maximum of 89.00, while the mean complexity of non mocked classes is 16.42 (max 420). Difference is also negligible (Wilcoxon p-value$=5.945e^{-07}$, Cliff's delta$=-0.166$).

We conjecture that the chosen code metrics are not enough to predict whether a class should be mocked. Future research needs to be conducted to understand how code metrics are related to mocking decisions.

There are many other discoveries that can be verified using quantitative studies (*i.e.* are slow tests mocked more often? how faster are test that use mocks?). Here we simply proposed an initial analysis to show its feasibility. Further research can be designed and carried out to devise approaches to quantitatively evaluate mocking practices.

VI. Related Work

Despite the widespread usage of mocks, very few studies analyzed current mocking practices. Mostafa et al. [32] conducted an empirical study on more than 5,000 open source software projects from GitHub, analyzing how many projects are using a mocking framework and which Java APIs are the most mocked ones. The result of this study shows that 23% of the projects are using at least one mocking framework and that Mockito is the most widely used (70%).

Marri et al. [29] investigated the benefits of using mock objects. The study identifies the following two benefits: 1) mock objects enable unit testing of the code that interacts with external APIs related to the environment such as a file system, and 2) enable the generation of high-covering unit tests.

Taneja et al. [40] stated that automatic techniques to generate tests face two significant challenges when applied to database applications: (1) they assume that the database that the application under test interacts with is accessible, and (2) they usually cannot create necessary database states as a part of the generated tests. For this reasons they proposed an "Automated Test Generation" for Database Applications using mock objects, demonstrating that with this technique they could achieve better test coverage.

Karlesky et al. [25] applied Test-Driven Development and Continuous Integration using mock objects to embedded softwares, obtaining an order of magnitude or more reduction in software flaws, predictable progress, and measurable velocity for data-driven project management.

Kim et al. [26] stated that unit testing within the embedded systems industry poses several unique challenges: software is often developed on a different machine than it will run on and it is tightly coupled with the target hardware. This study shows how unit testing techniques and mocking frameworks can facilitate the design process, increase code coverage and the protection against regression defects.

Tim et al. [28] stated that using Mock Objects is the only way to unit test domain code that depends on state that is difficult or impossible to reproduce. They show that the usage of mocks encourages better-structured tests and reduces the cost of writing stub code, with a common format for unit tests that is easy to learn and understand.

VII. Conclusion

Mocking is a common testing practice among software developers. However, there is little empirical evidence on how developers actually apply the technique in their software systems. We investigated *how* and *why* developers currently use mock objects. To that end, we studied three OSS projects and one industrial system, interviewed three of their developers, surveyed 105 professionals, and discussed the findings with a main developer from the leading Java mocking framework.

Our results show that developers tend to mock dependencies that make testing difficult, *i.e.*, classes that are hard to set up or that depend on external resources. In contrast, developers do not often mock classes that they can fully control. Interestingly, a class being slow is not an important factor for developers when mocking. As for challenges, developers affirm that challenges when mocking are mostly technical, such as dealing with unstable dependencies, the coupling between the mock and the production code, legacy systems, and hard-to-test classes are the most important ones.

Our future agenda includes understanding the relationship between code quality metrics and the use of mocking as well as the role of software evolution in developers' mocking practices.

REFERENCES

[1] Alura. http://www.alura.com.br/. [Online; accessed 03-Feb-2016].
[2] EasyMock. http://easymock.org. [Online; accessed 03-Feb-2016].
[3] JMock. http://www.jmock.org. [Online; accessed 03-Feb-2016].
[4] Kotlin. https://kotlinlang.org. [Online; accessed 03-Feb-2016].
[5] Mock. https://github.com/testing-cabal/mock. [Online; accessed 03-Feb-2016].
[6] Mocker. https://labix.org/mocker. [Online; accessed 03-Feb-2016].
[7] Mockito. http://site.mockito.org. [Online; accessed 03-Feb-2016].
[8] MyBatis. http://www.mybatis.org/. [Online; accessed 03-Feb-2016].
[9] PowerMock. https://github.com/powermock/powermock. [Online; accessed 03-Feb-2016].
[10] Sonarqube. https://www.sonarqube.org/. [Online; accessed 03-Feb-2016].
[11] Spring Framework. https://projects.spring.io/spring-framework/. [Online; accessed 03-Feb-2016].
[12] To Mock or Not To Mock? Online Appendix. https://doi.org/10.4121/uuid:fce8653c-344c-4dcb-97ab-c9c1407ad2f0.
[13] VRaptor. https://www.vraptor.com.br/. [Online; accessed 03-Feb-2016].
[14] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
[15] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
[16] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.
[17] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock roles, objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246. ACM, 2004.
[18] S. Freeman and N. Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.
[19] P. Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, 2004.
[20] B. Hanington and B. Martin. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.
[21] F. Henderson. Software Engineering at Google. feb 2017.
[22] M. R. Hess and J. D. Kromrey. Robust Confidence Intervals for Effect Sizes: A Comparative Study of Cohen's d and Cliff's Delta Under Non-normality and Heterogeneous Variances. *American Educational Research Association, San Diego*, nov 2004.
[23] A. Hunt and D. Thomas. *Pragmatic unit testing in c# with nunit*. The Pragmatic Programmers, 2004.
[24] T. Kaczanowski. *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, 2012.
[25] M. Karlesky, G. Williams, W. Bereza, and M. Fletcher. Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns. In *Embedded Systems Conference Silicon Valley (San Jose, California) ESC 413, April 2007*. ESC 413, 2007.
[26] S. S. Kim. *Mocking embedded hardware for software validation*. PhD thesis, 2016.
[27] J. Langr, A. Hunt, and D. Thomas. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf, 2015.
[28] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2001.
[29] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. *AST*, 9:149–153, 2009.
[30] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, dec 1976.
[31] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.
[32] S. Mostafa and X. Wang. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *2014 14th International Conference on Quality Software*, pages 127–132. IEEE, oct 2014.
[33] A. J. Nederhof. Methods of coping with social desirability bias: A review. *European journal of social psychology*, 15(3):263–280, 1985.
[34] R. Osherove. *The Art of Unit Testing: With Examples in .NET*. Manning, 2009.
[35] G. Rugg. A rticle picture sorts and item sorts. *Computing*, 22(3), 2005.
[36] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.
[37] H. Samimi, R. Hicks, A. Fogel, and T. Millstein. Declarative Mocking Categories and Subject Descriptors. pages 246–256, 2013.
[38] D. Spadini, M. Aniche, A. Bacchelli, and M. Bruntink. *MockExtractor*. The tool is available at https://github.com/ishepard/MockExtractor.
[39] D. Spencer. Card sorting: a definitive guide. http://boxesandarrows.com/card-sorting-a-definitive-guide/, 2004.
[40] K. Taneja, Y. Zhang, and T. Xie. MODA: Automated Test Generation for Database Applications via Mock Objects. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 289, New York, New York, USA, 2010. ACM Press.
[41] E. Weyuker. Testing component-based software: a cautionary tale. *IEEE Software*, 15(5):54–59, 1998.
[42] F. Wilcoxon. Individual comparisons of grouped data by ranking methods. *Journal of economic entomology*, 39(6):269, 1946.
[43] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229. IEEE, 2008.