Mock objects for testing java systems

Why and how developers use them, and how they evolve

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

DOI
10.1007/s10664-018-9663-0

Publication date
2018

Document Version
Final published version

Published in
Empirical Software Engineering

Empirical Software Engineering (2019) 24:1461–1498
https://doi.org/10.1007/s10664-018-9663-0

Mock objects for testing java systems

Why and how developers use them, and how they evolve

Davide Spadini[1,2] · Maurício Aniche[1] · Magiel Bruntink[2] · Alberto Bacchelli[3]

Published online: 6 November 2018

Abstract
When testing software artifacts that have several dependencies, one has the possibility of either instantiating these dependencies or using mock objects to simulate the dependencies' expected behavior. Even though recent quantitative studies showed that mock objects are widely used both in open source and proprietary projects, scientific knowledge is still lacking on how and why practitioners use mocks. An empirical understanding of the situations where developers have (and have not) been applying mocks, as well as the impact of such decisions in terms of coupling and software evolution can be used to help practitioners adapt and improve their future usage. To this aim, we study the usage of mock objects in three OSS projects and one industrial system. More specifically, we manually analyze more than 2,000 mock usages. We then discuss our findings with developers from these systems, and identify practices, rationales, and challenges. These results are supported by a structured survey with more than 100 professionals. Finally, we manually analyze how the usage of mock objects in test code evolve over time as well as the impact of their usage on the coupling between test and production code. Our 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 (e.g., infrastructure-related dependencies) and 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. Their perceptions are confirmed by our data, as we observed that mocks mostly exist since the very first version of the test class, and that they tend to stay there for its whole lifetime, and that changes in production code often force the test code to also change.

Keywords Software testing · Mocking practices · Mockito · Empirical software engineering

Communicated by: Abram Hindle and Lin Tan

✉ Davide Spadini
D.Spadini@tudelft.nl

Extended author information available on the last page of the article.

1 Introduction

In software testing, it is common that the software artifact under test depends on other components (Runeson 2006). Therefore, when testing a unit (*i.e.* a class in object-oriented programming), developers often need to decide whether to test the unit and all its dependencies together (similar to an integration testing) or to *simulate* these dependencies and test the unit in isolation.

By testing all dependencies together, developers gain realism: The test will more likely reflect the behavior in production (Weyuker 1998). However, some dependencies, such as databases and web services, may (1) slow the execution of the test (Meszaros 2007), (2)be costly to properly set up for testing (Samimi et al. 2013), and (3) require testers to have full control over such external dependencies (Freeman and Pryce 2009). 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 (2016), EasyMock (2016), and JMock (2016) for Java, Mock (2016) and Mocker (2016) 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 (Henderson 2017; Mostafa and Wang 2014) and has provided initial evidence that using a mock object can ease the process of unit testing (Marri et al. 2009).

Given the relevance of mocking, technical literature describes how mocks can be implemented in different languages (Hamill 2004; Meszaros 2007; Freeman and Pryce 2009; Osherove 2009; Kaczanowski 2012; Langr et al. 2015).

However, how and why practitioners use mocks, what kind of challenges developers face, and how mock objects evolve over time are still unanswered questions.

We see the answers to these questions as important to practitioners, tool makers, and researchers. Practitioners have been using mocks for a long time, and we observe that the topic has been dividing practitioners into two groups: The ones who support the usage of mocks (*e.g.* Freeman and Pryce (2009) defend the usage of mocks as a way to design how classes should collaborate among each other) and the ones who believe that mocks may do more harm than good (*e.g.* as in the discussion between Fowler, Beck, and Hansson, well-known experts in the software engineering industry community (Fowler et al. 2014; Pereira 2014)). An empirical understanding of the situations where developers have been and have not been applying mocks, as well as the impact of such decisions in terms of coupling and software evolution, can be used to help practitioners adapt and improve their future usage. In addition, tool makers have been developing mocking frameworks for several languages. Although all these frameworks share the main goal, they take different decisions: As an example, JMock opts for strict mocks, whereas Mockito opts for lenient mocks.[1] Our findings can inform tool makers when taking decisions about which features practitioners really need (and do not need) in practice. Finally, one of the challenges faced by researchers working on automated test generation concerns how to simulate a dependency

[1]When mocks are strict, the test fails if an unexpected interaction happens. In lenient mocks, tests do not fail for such reason. In Mockito 1.x, mocks are lenient by default; in Mockito 2.x, mocks are lenient, and by default, tests do not fail, and warnings happen when an unexpected interaction happens.

(Arcuri et al., 2014, 2017). Some of the automated testing generation tools apply mock objects to external classes, but automatically deciding what classes to mock and what classes not to mock to maximize the test feedback is not trivial. Our study also provides empirical evidence on which classes developers mock, thus possibly indicating the automated test generation tools how to do a better job.

To this aim, we perform a two-phase study. In the first part, we analyze more than 2,000 test dependencies from three OSS projects and one industrial system. Then, we interview developers from these systems to understand why they mock some dependencies and they do not mock others. We challenge and support our findings by surveying 105 developers from software testing communities and discuss our results with a leading developer from the most used Java mocking framework. In the second phase, we analyze the evolution of the mock objects as well as the coupling they introduce between production and test code in the same four software systems after extracting the entire history of their test classes.

The results of the first part of our study show that classes related to external resources, such as databases and web services, are often mocked, due to their inherently complex setup and slowness. Domain objects, on the other hand, do not display a clear trend concerning mocking, and developers tend to do it only when they are too complex. Among the challenges, a major problem is maintaining the behavior of the mock compatible with the original class (i.e., breaking changes in the production class impact the mocks). Furthermore, participants state that excessive use of mocks may hide important design problems and that mocking in legacy systems can be complicated.

The results of the second part of our study show that mocks are almost always introduced when the test class is created (meaning that developers opt for mocking the dependency in the very first test of the class) and mocks tend not to be removed from the test class after they are introduced. Furthermore, our results show that mocks change frequently. The most important reasons that force a mock to change are (breaking) changes in the production class API or (breaking) changes in the internal implementation of the class, followed by changes solely related to test code improvements (refactoring or improvements).

The main contributions of this paper are:

1. A categorization of the most often (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, based on interviews with developers of the analyzed systems and an online survey (RQ_2).
3. A list of the main challenges when making use of mock objects in the test suites, also extracted from the interviews and surveys (RQ_3).
4. An understanding of how mock objects evolve and, more specifically, empirical data on when mocks are introduced in a test class (RQ_4), and which mocking APIs are more prone to change and why (RQ_5).
5. An open source tool, namely MockExtractor, that is able to extract the set of (non) mocked dependencies in a given Java test suite.[2]

This article extends our MSR 2017 paper 'To Mock or Not To Mock? An Empirical Study on Mocking Practices' (Spadini et al. 2017) in the following ways:

1. We investigate when mocks are introduced in the test class (RQ_4) as well as how they evolve over time (RQ_5).

[2]The tool is available in our on-line appendix (Spadini 2017) and GitHub.

2. Our initial analysis on the relationship between code quality and mock practices considers more code quality metrics (Section 5.2).
3. We present a more extensive related work section, where we discuss empirical studies on the usage of mock objects, test evolution and test code smells (and the lack of mocking in such studies), how automated test generation tools are using mock objects to isolate external dependencies, and the usage of pragmatic unit testing and mocks by developers and their experiences (Section 6).

2 Background: Mock Objects

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

Mocking objects are a standard technique in software testing used to simulate dependencies. Software testers often mock to exercise the component under test in isolation.

Mock objects are available in most major programming languages. As examples, Mockito, EasyMock, as well as JMock are mocking frameworks available for Java, and Moq is available for C#. Although the APIs of these frameworks might be slightly different from each other, they provide developers with a set of similar functionalities: the creation of a mock, the set up of its behavior, and a set of assertions to make sure the mock behaves as expected. Listing 1 shows an example usage of Mockito, one of the most popular mocking libraries in Java (Mostafa and Wang 2014). 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 the method `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 lines 10 and 11, the mock will answer method invocations with the values provided in step 2.

Typically, methods of mock objects are designed to have the same interface as the real dependency, so that the client code (the one that depends on the component we desire to mock) works with both the real dependency and the mock object. Thus, whenever developers do not want to rely on the real implementation of the dependency (*e.g.* a database),

```
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

they can simulate this implementation and define the expected behavior using the approach mentioned above.

2.1 Motivating Example

Sonarqube is a popular open source system that provides continuous code inspection.[3] 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 (2016), a Java library for accessing databases.

Four test units 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 in some 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 example reinforces the idea that deciding whether or not to mock a class is not trivial. Developers have different reasons which vary according to the context. In this work, we investigate patterns of how developers mock by analyzing the use of mocks in software systems and we examine their rationale by interviewing and surveying practitioners on their mocking practices. Moreover, we analyze data on how mocks are introduced and evolve.

3 Reseach Methodology

Our study has a twofold *goal*. First, we aim at understanding how and why developers apply mock objects in their test suites. Second, we aim at understanding how mock objects in a test suite are introduced and evolve over time.

To achieve our first goal, we conduct quantitative and qualitative research focusing on four software systems and address the following questions:

RQ$_1$: **What dependencies do developers mock in their tests?** When writing an automated test for a given class, developers can either mock or use a concrete instance of its dependencies. Different authors (Mackinnon et al. 2001; Freeman et al. 2004) 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 and how often they do it by means of manual analysis in source code from different systems.

[3] https://www.sonarqube.org/

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 light on important aspects on which researchers and practitioners can effectively focus next. Therefore, we investigate the main challenges developers face when using mocks by means of interviews and surveys.

To achieve our second goal, we analyze the mock usage history of the same four software systems and answer the following research questions:

RQ_4: **When are mocks introduced in the test code?** In this RQ, we analyze when mocks are introduced in the test class: Are they introduced together with the test class, or are mocks part of the future evolution of the test? The answer to this question will shed light on how the behavior of software testers and their testing strategies when it comes to mocking.

RQ_5: **How does a mock evolve over time?** Practitioners affirm that mocks are highly coupled to the production class they mock (Beck 2003). In this RQ, we analyze what kind of changes mock objects encounter after their introduction in the test class. The answer to this question will help in understanding the coupling between mocks and the production class under test as well as their change-proneness.

3.1 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 (Mostafa and Wang 2014).

We select three open source software projects (*i.e.* Sonarqube,[4] Spring,[5] VRaptor[6]) and a software system from an industrial organization we previously collaborated with (Alura[7]). Tables 1 and 2 detail the size of these projects, as well as their mock usage. In the following, we describe their suitability to our investigation:

Spring Framework. Spring provides extensive infrastructural support for Java developers; its core serves as a base for many other offered services, such as dependency injection and transaction management. The Spring framework integrates with several other external software systems, which makes an ideal scenario for mocking.

Sonarqube. Sonarqube is a quality management platform that continuously measures the quality of source code and delivers reports to its developers. Sonarqube is a database-centric application, as its database plays an important role in the system.

VRaptor. VRaptor is an MVC framework that provides an easy way to integrate Java EE capabilities (such as CDI) and to develop REST web services. Similar to Spring MVC, the framework has to deal frequently with system and environment dependencies, which are good cases for mocking.

[4] https://www.sonarqube.org/
[5] https://projects.spring.io/spring-framework/
[6] https://www.vraptor.com.br/
[7] http://www.alura.com.br/

Table 1 The studied sample in terms of size and number of tests (N=4)

Project	# of classes	LOC	# of test units	# of test units with mock
Sonarqube	5,771	701k	2,034	652
Spring framework	6,561	997k	2,020	299
VRaptor	551	45k	126	80
Alura	1,009	75k	239	91
Total	13.892	1.818k	4.419	1.122

Alura. Alura is a proprietary web e-learning system used by thousands of students. It is a database-centric system developed in Java. The application resembles commercial software in the sense that it serves a single business purpose and makes heavy use of databases. According to their team leader, all developers make intensive use of mocking practices.

3.2 RQs 1, 2, 3: Data Collection and Analysis

The research method we use to answer our first three research questions follows a mixed qualitative and quantitative approach, which we depict in Fig. 1: (**1**) We automatically collect all mocked and non-mocked dependencies in the test units of the analyzed systems, (**2**) we manually analyze a sample of these dependencies with the goal of understanding their architectural concerns as well as their implementation, (**3**) we group these architectural concerns into categories, which enables us to compare mocked and non-mocked dependencies among these categories, (**4**) we interview developers from the studied systems to understand our findings, and (**5**) we enhance our results in an online survey with 105 respondents.

1. Data collection To obtain data on mocking practices, we first collect all the dependencies in the test units of our systems performing static analysis on their test code. To this aim, we create MockExtractor (Spadini et al. 2017), a tool that implements the algorithm below:

1. We detect all test classes in the software system. As done in past literature (*e.g.* Zaidman et al. 2008), 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).

Table 2 The studied sample in terms of mock usage (N=4)

Project	# of mocked dependencies	# of not mocked dependencies	Sample size of mocked (CL=95%)	Sample size of not mocked (CL=95%)
Sonarqube	1,411	12,136	302	372
Spring framework	670	21,098	244	377
VRaptor	258	1,075	155	283
Alura	229	1,436	143	302
Total	2,568	35,745	844	1,334

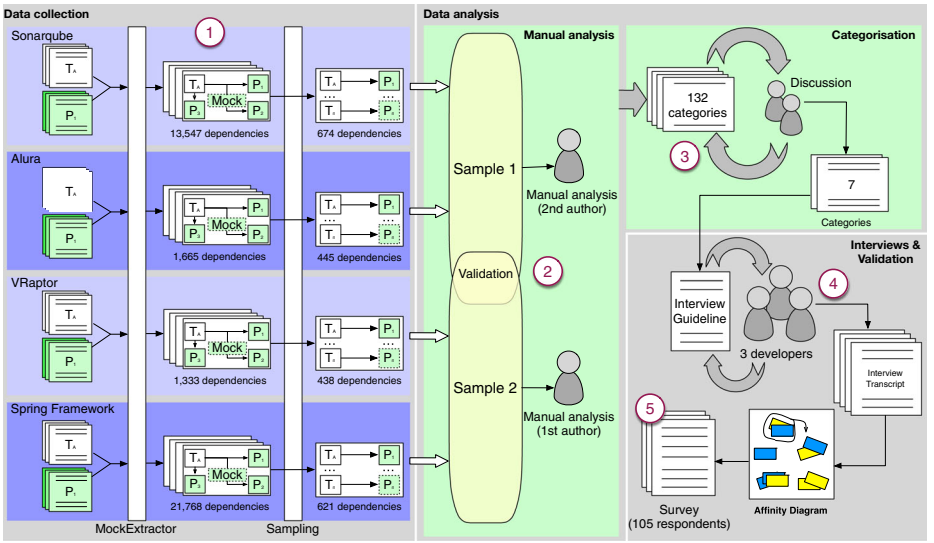

Fig. 1 The mixed approach research method applied

3. We mark each dependency as 'mocked' or 'not mocked.' Mockito provides two APIs for creating a mock from a given class:[8] (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 analyze 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, 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 (around 38,000), we analyze 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 create a sample for each project. We produce four samples, one belonging to each project. This gives us fine-grained information to investigate mock practices within each project.

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

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

The manual analysis procedure is as follows:

- Each researcher is in charge of two projects. The selection is made by convenience: The second author focuses on VRaptor and Alura, since he is already familiar with their internal structure.
- All dependencies in the sample are listed in a spreadsheet to which both researchers have access. Each row contains information about the test unit where the dependency was found, the name of the dependency, and a boolean indicating if that dependency was mocked.
- For each dependency in the sample, the researcher manually inspects the source code of the class. To fully understand the class' architectural concern, researchers can navigate through any other relevant piece of code.
- After understanding the concern of that class, the researcher fills the "Category" column with what best describes the concern. No categories are defined up-front. In case of doubt, the researcher first reads the test unit code; if not enough, he then talks with the other research.
- At the end of each day, the researchers discuss together their main findings and some specific cases.

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

3. Categorization To group similar categories we use a technique similar to card sorting (Rugg 2005): (**1**) each category is represented in a card, (**2**) the first two authors analyze the cards applying open (*i.e.* without predefined groups) card sort, (**3**) the researcher who created the category explain the reasons behind it and discuss a possible generalization (to make the discussion more concrete it is allowed to show the source code of the class), (**4**) similar categories are then grouped into a final, higher level category. (**5**) at the end, the authors give 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 represent some business logic of the system and has no external dependencies. The full list of the 116 categories is available in our on-line appendix (Spadini 2017).

4. Interviews We use the results from our investigation on the dependencies that developers mock (RQ_1) as an input to the data collection procedure of RQ_2. We design an interview in which the goal is to understand *why* developers did mock some roles and did not mock other roles. The interview is semi-structured and is conducted by the first two authors of this paper. For each finding in previous RQ, we ensure that the interviewee describes why they did or did 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 (Spadini 2017).

As a selection criterion for the interviews, we aimed at technical leaders of the project. Our conjecture was that technical leaders are aware of the testing decisions that are taken by the majority of the developers in the project. In practice, this turned out to be true, as our

interviewees were knowledgeable about these decisions and talked about how our questions were also discussed by different members of their teams.

To find the technical leaders, we took a different approach for each project: in Alura (the industry project), we asked the company to point us to their technical leader. For VRaptor and Spring, we leveraged our contacts in the community (both developers have participated in previous research conducted by our group). Finally, for Sonarqube, as we did not have direct contact with developers, we emailed the top 15 contributors of the projects. Out of the 15, we received only a single (negative) response.

At the end, we conduct three interviews with active, prolific developers from three projects. Table 3 shows the interviewees' details.

We start each interview by asking general questions about interviewees' decisions with respect to mocking practices. As our goal is to explain the results we found in the previous RQ (the types of classes, e.g., database and domain objects, as well as how often each of them is mocked by developers), we present the interviewee with two tables: one containing the numbers of each of the six categories in the four analyzed projects (see RQ_1 results, Fig. 3), and another containing only the results of the interviewee's project.

We do not show specific classes, as we conjecture that remembering a specific decision in a specific class can be harder to remember than the general policy (or the "rule of thumb") that they apply for certain classes. Throughout the interview, we reinforce that participants should talk about the mocking decisions in their specific project (which we are investigating); divergent personal opinions are encouraged, but we require participants to explicitly separate them from what is done in the project. To make sure this happens, as interviewers, we question participants whenever we notice an answer that did not precisely match the results of the previous RQ.

As aforementioned, for each category, we present 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 ask:

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 is in charge of summarizing the answers. Before finalizing the interview, we revisit the answers with the interviewee to validate our interpretation of their opinions. Finally, we close the interview by asking questions about the current challenges they face when applying mock practices in their projects.

Table 3 Profile of the interviewees

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

Interviews are conducted via Skype and fully recorded, as well as manually transcribed by the researchers. With the full transcriptions, we perform card sorting (Spencer 2004; Hanington and Martin 2012) to identify the main themes.

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

5. Survey To challenge and expand the concepts that emerge during the previous phases, we conduct a survey. All questions are derived from the results of previous RQs. The survey has four main parts. (1) In the first part, we ask respondents about their experience in software development and mocking. (2) The second part of the survey asks respondents about how often they make use of mock objects in each of the categories found during the manual analysis. (3) The third part asks respondents about how often they mock classes in specific situations, such as when the class is too complex or coupled. (4) The fourth part focuses on challenges with mocking. Except for the last question, which is open-ended and optional, the questions are closed-ended and based on a 5-point Likert scale.

We initially design the survey in English, then we compile a 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 analyze the open questions by performing card sorting. The full survey can be found in our on-line appendix (Spadini 2017).

We received a total of 105 answers from both Brazilian Portuguese and English surveys. The demographics of the participants can be found in Fig. 2. 22% of the respondents have between one and five years of experience, 64% between 6 and 15 and 14% have more than 15 years of experience. The most used programming languages are Java (52%), JavaScript (42%), and C# (39%). Among the respondents, the most used mocking framework is Mockito (53%) followed by Moq (31%) and Powermock (8%). Furthermore, 66% of the participants were from South America, 21% from Europe, 8% from North America, and the remaining 5% from India and Africa. Overall our survey reached developers from different experience levels, programming languages, and mocking framework.

3.3 RQ_4 and RQ_5: Data Collection and Analysis

To understand how mock objects evolve over time and what type of changes developers perform on them, we (i) collect information about test classes that make use of mocks and (ii) manually analyze a sample to understand how and why mocking code changes.

1. Data extraction We extract information about (1) when mock objects are introduced in test classes and (2) how the mocking code changes over time. To this aim, we create a static analysis tool and mine the history of the four analyzed systems. The tool implements the algorithm below:

For the mock introduction:

1. For each class in a commit, we identify all test classes. As done for MockExtractor, we consider a class to be a test when its name ends with `Test` or `Tests`. For each test class, we check if it makes use of at least one mock object, by checking whether the class imports Mockito dependencies.

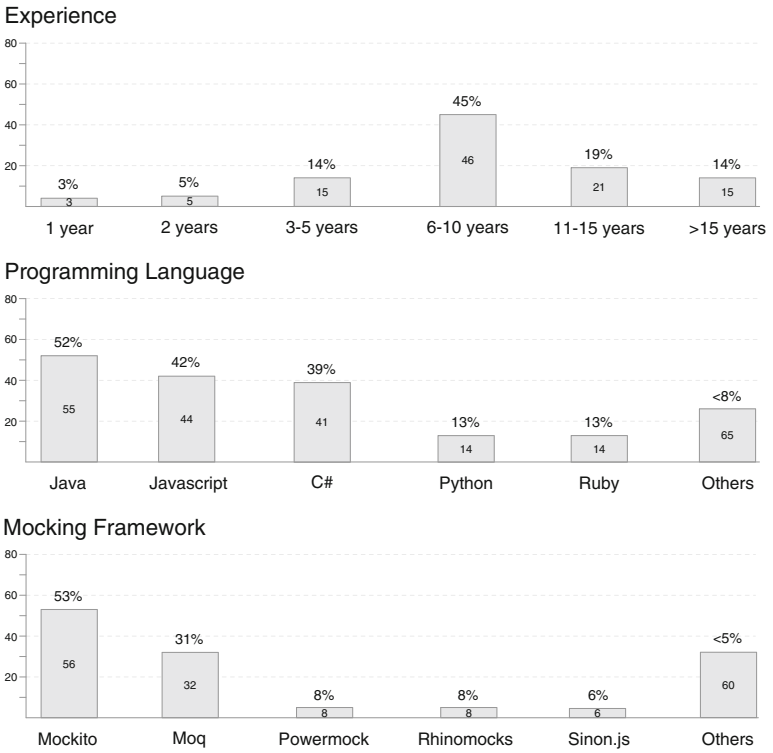

Fig. 2 High-level survey participant details (n=105). Figures on Programming language and Mocking framework are higher than 100%, as participants could choose multiple options

2. Given a test class and an indication of whether that test class makes use of mocks, we classify the change as follows:
   (a) If the test class contains mocks and the test class is new (*i.e.* Git classifies this modification as an addition), we consider that mocks were *introduced from its creation*.
   (b) If the test class does not contain mocks, but it previously did, we consider that *mocks were removed* from that test class.
   (c) If the test class contains mocks and it is a modification of a test:
      (i) If the test did not contain mocks before this change, we consider that mocks were *introduced* later in the test class' lifespan.
      (ii) If the test contained mocks before this change, we keep the current information we have about this class.

For the mock evolution:

1. For each test class that uses at least one mock object, we check whether developers modify any code line involving a mock. Since Mockito provides many APIs (Mockito 2016), we keep track only of the most used ones (Mostafa and Wang 2014). The APIs that we considered are depicted in Table 4. First, we obtain the list of modified lines of each test that involve Mockito APIs. Then, for each line:
   (a) We check the type of change, namely addition, deletion, or modification. To capture the latter, we adopt a technique similar to the one proposed by Biegel et al. (2011),

Table 4 The Mockito APIs we study the evolution in the four software systems

Mockito API	Provided functionality
verify()	Verifies certain behavior happened once.
when()	Enables stubbing methods.
mock()	Creates a mock with some non-standard settings.
given()	Syntax sugar for BDD (Wynne and Hellesoy 2012a) practitioners. Same as when().
doThrow()	Stub the void method with an exception.
doAnswer()	Stub a void method with generic Answer.
doNothing()	Setting void methods to do nothing.
doCallRealMethod()	Call the real implementation of a method.
doReturn()	Similar to when(). Use only in the rare occasions when you cannot use when().
verifyZeroInteractions()	Verifies that no interactions happened on given mocks beyond the previously verified interactions.
verifyNoMoreInteractions()	Checks if any of given mocks has any unverified interaction.
thenReturn()	Sets a return value to be returned when the method is called.
thenThrow()	Sets a Throwable type to be thrown when the method is called.

based on the use of textual analysis. Specifically, if the cosine similarity (Baeza-Yates et al. 1999) between two lines in the diff of the previous version of the file and the new version of the file is higher than α, then we consider the two lines as a modification of the same line.

(b) We obtain the list of Mockito APIs involved in the line using regular expressions.

(c) Depending on the type of action we discover before, we update the number of times the corresponding Mockito API was added, deleted or changed.

To determine the α threshold, we randomly sample a total of 75 real changes from the four software systems and test the precision of different thresholds, from 1% to 100%. Based on this process, we choose $\alpha = 0.71$ (which leads to a precision of 73%).

2. Manual analysis The goal of the analysis is to understand what drives mock objects to change once they are in a test class. To that aim, we manually analyze changes in the mocks extracted in the previous step. We opt for manual analysis as it is necessary to understand the context of the change.

Due to the size of the total number of modified lines involving Mockito APIs (∼10,000), we analyzed a random sample. Similarly to our previous manually analysis, to obtain more fine-grained information, we create a sample of 100 changes for each project. This sample gives us a confidence level of 95% and an error (E) of 10%.

The manual analysis procedure is as follows:

- All the mock changes in the sample are listed in a spreadsheet. Each row contains information about the commit that modifies the line (*i.e.* the commit hash), and how the line changed (its previous and successive versions).
- For each mock that changed, researchers manually inspect the change, with the goal of understanding the reasoning behind the change, *i.e.* why the mock changed. To fully

understand why the change happened, researchers also inspect the respective commit and the changes involved in there. This step gives information not only about the change in the mock itself, but also about the possible changes in production class being mocked, the test class, and the class under test. The researchers are not aware of the details of the project, and thus, they are not able to explain the change from the business perspective of the project (e.g., a mock changed due to a new feature that is introduced); rather, researchers focus on understanding whether the change was caused either because the test changed or because the production code changed and forced the mock to change together.
- After understanding all the context of the change and the reason behind the mock being changed, the researcher attributes a code (*i.e.* reason) that best describes the change. As done for the first part of our study, no categories are defined up-front.
- The first 20 changes are done together by the two researchers so that both could adapt to the process. After, each researcher is in charge of two projects. During the entire analysis, researchers discuss odd cases as well as share and iteratively refine their code book.

4 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, the challenges developers face in this context, as well as the introduction and evolution of mocks.

4.1 RQ_1: What Dependencies Do Developers Mock in Their Tests?

As visible in Table 1, 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.

Since 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. We obtain a total of 11,824 non-mocked and 938 mocked dependencies. The intersection of these two sets reveals that 650 dependencies (70% of all dependencies mocked at least once) were both mocked and non-mocked in the test suite.

In Fig. 3, 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 appear more frequently than other types of external dependencies. In the following, we detail 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 with the definition of Domain Object (Evans 2004) and Domain Logic (Fowler 2002) 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 (Fowler 2002) pattern).

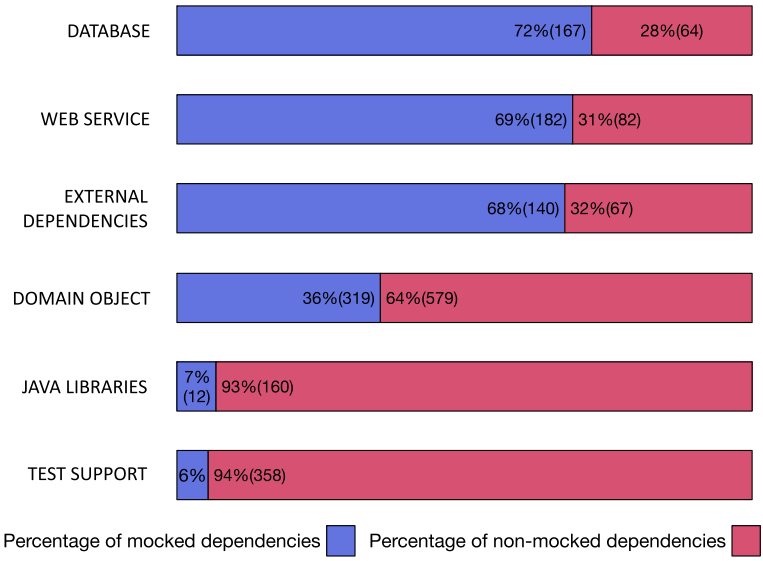

Fig. 3 How often each architectural role is (not) mocked in the analyzed systems ($N = 2,178$)

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.

Unresolved: Dependencies that we were not able to solve. For example, classes belonging to a sub-module of the project for 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%).[9]

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 (Mackinnon et al. 2001; Hunt and Thomas 2004), 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.

[9]We present the numbers for each project in our online appendix (Spadini 2017).

> RQ_1. Classes that deal with external resources, such as databases and web services, are often mocked. There is no clear trend for domain objects.

4.2 RQ_2. 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 3.

Mocks are 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 an 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 Fig. 4, 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 want to make sure that my SQL works. Other than that, we mock."* This is also confirmed in our survey (Fig. 4), as our respondents also almost never mock the class under test.

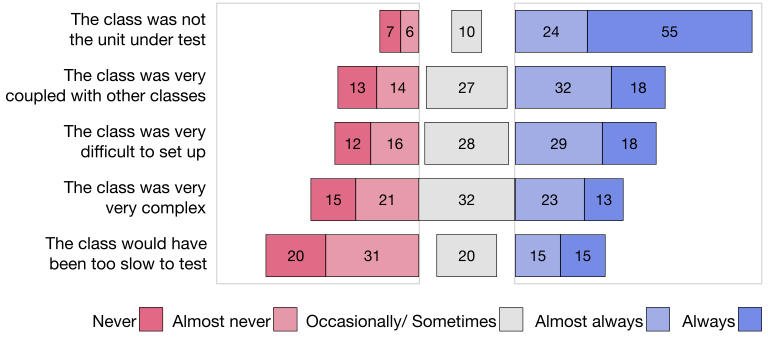

Fig. 4 Reasons to use mock objects ($N = 105$)

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 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, making an HTTP Request] I create an interface that represents the side effect and [instead of using a specific implementation] I mock the interface directly."*

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 5 shows that answers about mocking 'Domain objects' vary. There is a slight trend towards not mocking them, in line to our findings during the interviews and in RQ_1.

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,

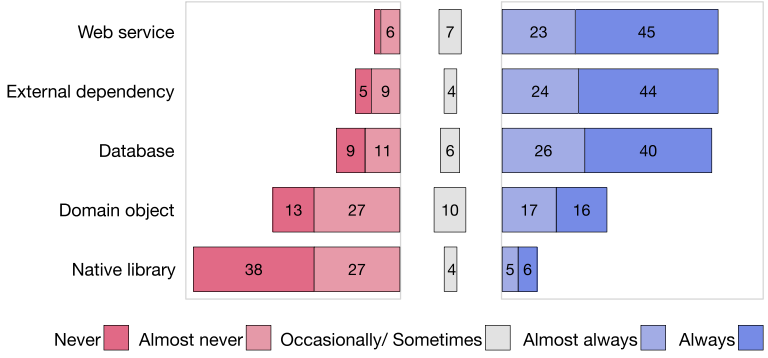

Fig. 5 Frequency of mocking objects per category ($N = 105$)

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 match with technical literature (Mackinnon et al. 2001; Hunt and Thomas 2004).

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 Fig. 5, we observe that respondents always or almost always mock 'Web services' (∼82%), 'External dependencies' (∼79%) and 'Databases' (∼71%). This result confirms 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 using the concrete implementation would be problematic (*e.g.*, the class would be too slow or complex to set up).

4.3 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 survey question about challenges (which received 61 responses). Categories 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 the 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."*

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 (2016) (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 indicator of poorly engineered code. During the interviews, D1, D2, and D3 mentioned *the same example* where using mocks can hide a deeper problem in the system's design: indeed, they all said that a developer could mock a class with a lot of dependencies (to ease testing), but the problem would remain, since a class with a lot of dependencies probably represents a design flaw 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 poorly 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 the original class, especially when the original 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, such as maintaining the behavior of the mock compatible with the original class, the relationship between how much mocking is necessary is to test a class and its code quality, and the (not positive) excessive use of mock objects to test legacy systems.

4.4 RQ_4. When are Mocks Introduced in the Test Code?

In Table 5, we summarize the information about the introduction of mock objects in the four studied systems. We observe that:

– The vast majority of mocks (2,159, or 83% of the cases) are introduced at the inception of the test class. In a minority of cases (433, or 17% of the cases), mocks are introduced later in the lifetime of the test class. These results are consistent across the four studied systems, with an approximate 80/20 ratio. These results seem to indicate that developers

Table 5 When mock objects were introduced (N=2,935)

	Spring	Sonarqube	VRaptor	Alura	Total
Mocks introduced from the beginning	234 (86%)	1,485 (84%)	177 (94%)	263 (74%)	2,159 (83%)
Mocks introduced later	37 (14%)	293 (16%)	12 (6%)	91 (26%)	433 (17%)
Mocks removed from the tests	59 (22%)	243 (14%)	6 (3%)	35 (10%)	343 (13%)

generally tend to *not* refactor test cases to either introduce mocks or to delete them, instead they use mocks mostly for new tests. We hypothesize that this behavior may hint at the fact that developers (i) do not consider important to refactor their test code (as found in previous research (Zaidman et al. 2008), although this behavior could lead to critical test smells (van Deursen et al. 2001)), (ii) do not consider mocks a good way to improve the quality of existing test code, or (iii) start with a clear mind on whether they should use mocks to test a class. Investigating these hypotheses goes beyond the scope of this work, but studies can be devised to understand the reasons why developers adopt this behavior.

– Of these mocks, 343 (13%) were removed afterward (we consider both the cases in which the mock is introduced from the inception of the test class and the cases when the mock is introduced in a later moment). To have a more clear idea of what these removals correspond to, we manually inspect 20 cases. We observe that, in most cases, developers replace the mock object with the real implementation of the class. Despite this manual analysis, since we are not developers of the system, it is hard to pinpoint the underlying reason for the changes that convert tests from using mocks to using the real implementation of a class. Nevertheless, it is reasonable to hypothesize that the choice of deleting a mock is influenced by many different factors, as it happens for the choices of (not) mocking a class, which we reported in the previous sections.

> ***RQ*$_4$.** In the studied systems, mocks are mostly (80% of the time) present at the inception of the test class and tend to stay in the test class for its whole lifetime (87% of the time).

4.5 RQ$_5$. How Does a Mock Evolve Over Time?

Table 6 shows the evolution of the Mockito APIs over time. For each API, we have three categories: 'added,' 'changed,' and 'removed.' These categories correspond to the situations in which the lines containing the calls to the APIs are modified; we map the categorization of added/changed/removed as it provided by git. Furthermore, Table 6 reports the statistics for each project and a summary in the last column.

As expected[10] the API calls `verify()`, `when()`, `mock()` and `thenReturn()` are the most used ones. These results are complementing previous studies that reported similar findings (Mostafa and Wang 2014). Furthermore, we note that in Sonarqube developers make intense use of mock objects, especially by using the `when()` and `thenReturn()` APIs. Spring Framework is the only project that uses the `given()` API, which belongs to

[10] In fact, it is not possible to correctly use Mockito without these API calls.

Table 6 The evolution of the Mockito APIs over time (N=74,983)

Mockito API	Spring framework	Sonarqube	VRaptor	Alura	Total
Added calls to API					
verify()	3,767	5,768	685	857	11,077
when()	533	9,531	1,771	1,961	13,796
mock()	1,316	6,293	334	590	8,533
given()	1,324	0	0	9	1,333
doThrow()	42	139	57	5	243
doAnswer()	1	13	1	0	15
doNothing()	11	15	3	0	29
doReturn()	5	111	19	24	159
verifyZeroInteractions()	22	556	15	35	628
verifyNoMoreInteractions()	115	605	3	6	729
thenReturn()	454	8,602	1,671	1,850	12,577
thenThrow()	8	135	30	12	185
Changed calls to API					
verify()	316	2,253	53	207	2,829
when()	53	2,731	134	386	3,304
mock()	121	1,804	6	71	2,002
given()	223	0	0	0	223
doThrow()	0	26	1	0	27
doAnswer()	0	5	0	0	5
doNothing()	0	0	0	0	0
doReturn()	0	10	0	1	11
verifyZeroInteractions()	0	83	1	5	89
verifyNoMoreInteractions()	4	61	2	0	67
thenReturn()	30	1,162	113	160	1,465
thenThrow()	1	10	7	0	18
Deleted calls to API					
verify()	646	2,141	110	250	3,147
when()	125	3,651	327	607	4,710
mock()	221	2,397	87	206	2,911
given()	136	0	0	1	137
doThrow()	9	31	4	0	44
doAnswer()	0	2	1	0	3
doNothing()	0	11	3	5	19
doReturn()	3	37	7	5	52
verifyZeroInteractions()	8	141	3	2	154
verifyNoMoreInteractions()	11	162	1	2	176
thenReturn()	113	3,269	309	557	4,248
thenThrow()	1	31	4	3	39

BDD Mockito, a set of APIs for writing tests according to the Behavior Driven Development process (Wynne and Hellesoy 2012b).

The most deleted APIs are `verify()`, `when()`, `mock()`, and `thenReturn()`. This is expected since these are the most frequently used APIs, thus it is more likely that they are later deleted. However, as to why they are deleted, reasons may be many. We see two plausible explanations, also taking into account the results from RQ_4: The first explanation for deletions is that the developer decided to delete the test, maybe because obsolete or not useful anymore; the second explanation is instead that the developer decided to replace the mock object with the real implementation of the production class.

Turning our attention to the Mockito APIs that change the most ('Changed calls to API' in Table 6), Table 7 shows the results of our manual analysis on a sample of more than 300 of these changes. We found that there are mainly two reasons for mocks to change: (a) the production code induced it ($113 + 59 = 172$, or 51%) or (b) improvements to the test code triggered it (164, or 49%). We detail these two cases in the following:

a) Mocks that changed due to changes in the production code. With our analysis, we observe two different types of changes that happen in production code that induce mocks to change: 'changes in the production class API' and 'changes in the internal implementation of the class.'

The former happens when the API of the production class being mocked changes in any way: the return type of the method is changed (13% of the cases), the number of parameters received by the method is changed (27%), the method is renamed (30%), or the entire class is either renamed or refactored (30%).

As discussed during the interviews with developers, a major challenge using mocks is correctly handling the coupling between production and test code. Our manual analysis corroborates the presence of this challenge: Indeed, we note how a change—even if minor, such as a renaming—in a production class can affect *all* its mocks in the test code. Interestingly, Sonarqube is the project in which this happens the most.

Concerning 'changes in the internal implementation of the class,' we found cases in which developers changed the internal encapsulated details of how a method works from the inside and this induced the mock to change accordingly. We observe this phenomenon in 73% of cases. Besides, we also observe classes moving away from a method and replacing it for another one (19%) and even production classes being completely replaced by others (8%), which then led to changes in the way the class/method is mocked in the test.

b) Mocks that changed during test evolution. Changes in the test code itself, not related to production code, are one of the main reason for mock objects to change. Among all the changes related to test code, we find that Mockito APIs change because of test

Table 7 Classification of the changes to Mockito APIs

Type of change	Spring framework	Sonarqube	VRaptor	Alura	Total
Test code related	57	20	42	45	164
Production class' API	19	33	36	25	113
Production class' internal implementation details	8	22	15	14	59

refactoring (63%). More specifically, we observed mocks being changed due to the field or variable that hold their instances to be renamed, the *static import* of the Mockito's API so that the code becomes less noisy, and general refactoring on the test method. We also observed mocks being changed due to improvements that developers make in the test itself (32%), *e.g.* testing different inputs and corner cases.

Interestingly, test refactorings that involve mocks happen more often in Alura, Spring, and VRaptor than in Sonarqube.

Finally, these results are in line with what we observed in RQ_3: Developers considered unstable dependencies, *i.e.* maintaining the behavior of the mock compatible with the behavior of the original class, as a challenge.

> ***RQ*$_5$.** Lines involving mocks change often. Test refactoring (not related to the mocks themselves), changes to the mocked production class, and changes to the internal implementation of the mocked class, are the most frequent reasons that induce a mock to change.

5 Discussion

In this section, we present the results of a debate about our findings with a core developer from Mockito. Next, we provide an initial quantitative evaluation of the mocking practices that emerged in our results and how much they apply to the systems under study. Finally, we discuss the main findings and their implications for both practitioners and future research.

5.1 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 (Fig. 3), he stated: *"If someone tells us that s/he is spending 100 boiler-plate 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 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 explained that Mockito developers have a philosophical debate internally: They want to keep a clear line of what this framework should and should not do. Not supported features such as the possibility of mocking a static method would enable developers to test their legacy code more efficiently. 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.”* Indeed, mock proponents often believe that making use of static methods is a bad practice. Static methods cannot be easily mocked. In Java, mock frameworks dynamically create classes at runtime that either implement an interface or inherit from some base class, and implement/override its methods. Since it is impossible to override static methods in Java, a mock framework that wants to support such feature would have to either modify the bytecode of a class at runtime or replace the JVM’s default classloader during the test execution.[11] As a consequence, static methods cannot be easily replaced by a mock implementation during a test; therefore, whenever a class invokes a static method developers have less control on their tests (regardless of whether this static method is part of the class under test or of an external class). Because of this limitation, mock proponents often suggest developers to write wrappers around static methods to facilitate testing (*e.g.* a class `Clock` containing a `now()` instance method that wraps Java’s `Calendar.getInstance()`).[12]

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 (Kotlin 2016), 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 of how to better use the framework. However, according to him, knowing what should be mocked and what should not be mocked comes with experience.

5.2 Relationship Between Mocks and Code Quality

A recurrent topic throughout our interviews and surveys was about a possible relationship between the usage of mocks and the code quality of the mocked class. In other words, when classes are too coupled or complex, developers might prefer to mock their behavior instead of using their concrete implementation during the tests.

In this section, we take a first step towards the understanding of this relationship, by means of analyzing the code quality metrics of mocked/not mocked classes.

We take into account four metrics: CBO (Coupling between objects), McCabe’s complexity (McCabe 1976), LOC (Lines of Code), NOM (Number of methods). We choose these metrics since they have been widely discussed during the interviews and, as pointed out during the surveys, developers mock when classes are very coupled or difficult to set up. In addition, CK metrics have proven to be useful in different predicting tasks, such as bug prediction (D’Ambros et al. 2010) and class testability (Bruntink and Van Deursen 2004).

[11] As an example, Powermock (a Java framework that can mock static methods) makes use of both bytecode manipulation and a custom classloader. More information can be found at the project’s official page: https://github.com/powermock/powermock. Last access in July, 2018.

[12] The Clock wrapper example is taken from the Alura project.

To obtain software code quality metrics, we used the tool CK[13]: we chose this tool because (i) it can calculate code metrics in Java projects by means of static analysis (i.e., no need for compiled code), and (ii) authors were already familiar with this tool (Aniche et al. 2016).

We linked the output of our tool MockExtractor and CK: for each production class in the four systems, we obtained all the necessary code metrics and the number of times the class was/not mocked. With the metrics value for each production class, 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').

Furthermore, to control for the fact that classes may be tested a different number of times (which could influence the developer's mocking strategy), we divided the classes into four categories, according to the number of times they were tested. To choose the categories, we used the 80^{th}, 90^{th} and 95^{th} percentiles, as done by previous research (Alves et al. 2010; Aniche et al. 2016). The used thresholds where: Low ($x \leq 4$), Medium ($4 < x \leq 7$), High ($7 < x \leq 12$), and Very High ($x \geq 12$).

To compare code metrics of mocked and not mocked production classes, we use the Wilcoxon rank sum test (Wilcoxon 1946) (with confidence level of 95%) and Cliff's delta (Hess and Kromrey 2004) to measure the effect size. We choose Wilcoxon because our distribution is not normal (we checked by both inspecting the histogram as well by running Shapiro-Wilk test (Razali and Wah 2011)) and because it is a non-parametric test (does not have any assumption on the underlying data distribution).

In Fig. 6, we show the results of the comparison between code metrics in mocked/not mocked classes. For every metric, we present the difference between mocked and not mocked classes in the 4 categories. To better present the results, we use a log scale on the *y*-axis.

As a result, we see that both mocked and non mocked classes are similar in all the metrics. From Wilcoxon rank sum test and the effect size, we observed that almost in all the cases the overall difference is negligible (Wilcoxon p-value$<$0.001, Cliff's Delta<-0.12). There are however some exceptions: in terms of LOC, we can notice that in high and very high tested classes, the value for the mocked ones is slightly higher (Cliff's Delta=-0.37). Regarding complexity, again for high and very high tested classes, the value for the mocked category is higher than for the non mocked category (Cliff's Delta=-0.29).

Interestingly, these results *are not* in line with what we discovered during the interviews and surveys: indeed, even though developers said that they prefer to mock complex or highly coupled classes, it seems not to be the case if we look at code metrics. We conjecture that the chosen code metrics are not enough to explain when a class should or should not be mocked. Future work should better understand how code metrics are related to mocking decisions.

5.3 To Mock or Not to Mock: the Trade-Offs

We started this paper by stating the trade-off that developers have to make when deciding to use mocks in their tests: *by testing all dependencies together, developers gain realism.*

[13] https://github.com/mauricioaniche/ck

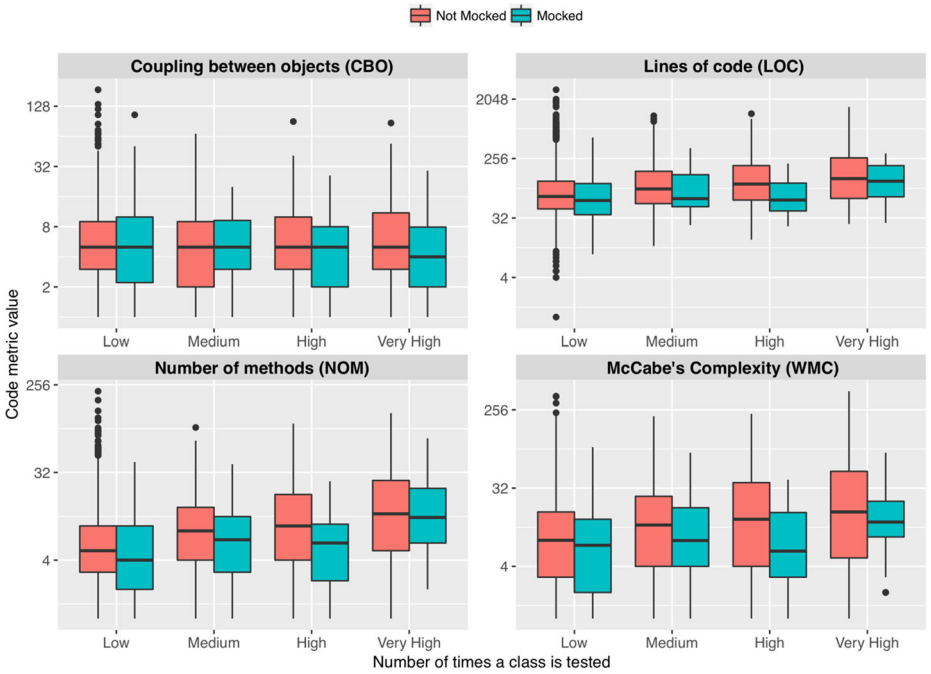

Fig. 6 Code quality metrics comparison between mocked/not mocked classes (log scale)

By simulating its dependencies, developers gain focus. Moreover, our first paper was called "To Mock or Not to Mock?".

Indeed, our study has been around the decision process that developers go through when deciding whether to use mocks in their tests. As with most decisions in software engineering, deciding whether to use mocks comes with pros and cons. The trade-offs in these different decisions were an orthogonal topic throughout our research findings. In this section, we group and highlight them.

Firstly, our interviewees (technical leaders) were all aware that their tests become less realistic with mocks and that, as a consequence, (an important) part of their system was not being actually tested. To tackle this problem, all the interviewees have been combining different levels of testing in their test suites: A class might be mocked throughout the tests suites to enable other classes to be unit tested; however, that same class, when being the class under test, is tested via integration tests. As an example, the Alura technical leader explained that their Data Access Objects (DAOs) are often mocked throughout their test suite; however, all their DAOs have dedicated integration tests that aim at exercising the SQL query in a real external database. We made the same observation when exploring Sonarqube's test suite.[14]

This is a trade-off they have been making: they mock a certain dependency, so that tests are easier to be written. However, they pay the price of writing integration tests for these dependencies later on, to make sure these dependencies really work as expected.

[14]The `org.sonar.db.user.UserDao` is an example of such class. The DAO is mocked throughout the test suite, and the DAO itself is tested by means of an integration test (see `UserDaoTest` class).

Secondly, the interviewees made effort to have their systems "easy to be tested". This can mean two different things: designing classes in such a way that mocking is possible, and avoiding complex classes that would require too much mocking. For the former, we see how developers have been taking class design decisions "for the sake of testability". The Spring developer affirmed to create interfaces that represent side effects, such as HTTP calls (which are naturally harder to be tested in isolation), just for the sake of simplifying the test. As exemplified before, Alura has a "Clock" abstraction to ease the simulation of different dates and times. The same pattern happens in Sonarqube[15]. This is another trade-off developers are making when it comes to testability and mocking: on one hand, the system gets more complex (in fact, new abstractions are introduced to the code base), on the other hand writing tests gets easier.

Concerning the latter, we also observed how simplicity plays a great role in how interviewees judge the difficulty of testing a class. For example, when our interview comes to the point where we discussed domain objects, we could see that all interviewees only thought mocking this type of classes when too complex. In addition, while we could not observe any relationship between CK metrics and the usage of metrics, an important perception of our participants was that complexity means difficulty in testing, which implies in the (possibly excessive) usage of mocks.

Finally, our interviewees are aware of the coupling they introduce when using mock objects and the fragility this brings to their test suites. In other words, they know that whenever they make use of a mock, that mock might be sensitive to changes in the original class. Such changes are naturally propagated to their test suites, which will then require developers to spend time in fixing them. Apparently, this is a trade-off they currently choose to pay: more testability comes with the price of a higher fragility in their test code.

Overall developers are aware of the positive and the negative aspects of using mock objects in their test suites. The question we raise for future researchers is: *what we can do to reduce the impact of the negative ones?*

5.4 Implications for Developers, Tool Makers, and Researchers

Mocking is a popular topic among software developers. Due to its importance, different researchers and practitioners have been writing technical literature on mock objects (*e.g.* Hamill 2004; Meszaros 2007; Freeman and Pryce 2009; Osherove 2009; Kaczanowski 2012; Langr et al. 2015), ranging from how to get started with mocks to best practices. The need for empirical studies on mocking practices was previously mentioned by Xie et al. (2010). According to them, understanding what types of components developers mock and its impact is essential for the future of developer testing. Our research complements such literature in several 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 (2007) 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).

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

[15] See `PurgeProfiler` and its `Clock` internal class.

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 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 (Beck 2003).

In addition, our study provides evidence that the coupling between the production code being mocked and test code indeed exists and may impact the maintenance of the software system. As our results show, around 50% of changes in mock objects are actually due to implementation changes in production classes, *e.g.* the signature or the return type of a method changed, and the developer had to fix the mock. In practice, this means that developers are often required to fix their test code (more specifically, the mocks in these test classes) after changing production classes. It is important to notice that such problem would not have happened if test classes were not making use of mock objects; after all, when a test makes use of the concrete implementation of a dependency instead of a mock, there is no need for the test to know how the production class under test will use this dependency. As a consequence, the test class is less coupled to the dependency. This paper not only help developers to decide whether mocks are a valid option for that test, but also paves the way for future work on how to reduce the inherent coupling that mocks introduce to the test code.

Finally, the empirical knowledge on the mocking practices of developers can be useful to tool makers as it (1) provides more awareness on how mocks are used as well as on the possibly problematic coupling between test and production code, and it (2) eases the mocking of similar infrastructure-related dependencies.

More specifically, to the former, we see tool makers proposing ways to warn developers about the usage and the impact their mocks (*e.g.* "this dependency is often mocked" or "X mocks would be affected in this production changes"). Moreover, we raise the question on whether mocking APIs could be done in such a way that the existing (and currently strong) coupling between test and production code would be smaller.

To the latter, we foresee tool makers proposing tools that would help developers in mocking infrastructure-related code (*e.g.* database access, file reading and writing), as they are the most popular types of mocked dependencies. A developer who mocks a database dependency will likely spend time simulating common actions, such as "list all entities" and "update entity". A tool could spare the time developers spend in creating repeated simulations for similar types of dependencies (*e.g.* DAOs share many similarities in common).

5.5 Threats to Validity

In this section, we pose possible threats to the validity of our results as well as the actions we took to mitigate them.

Construct validity Threats to *construct validity* concern our research instruments:

1. 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 "not 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.

2. In the first part of the study, as only a single researcher manually analyzes each class and there could be divergent opinions despite the discussion mentioned above, 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 Fig. 1, Point 2. The final agreement on the seven categories was 89%.
3. In the second part of the study, namely the evolution of mocks, we devised a tool that extracts source code information about the changes that mock objects suffer throughout history. To keep track of the changes, we had to link source code lines in the old and new version of the *source code diff* that Git provides between two commits; such link is not readily available. To that aim, as we explain in Section 3, we apply cosine similarity to determine whether two lines are the same. Such heuristic may be prone to errors. In order to mitigate this thread, we determined the threshold for the cosine similarity after experimenting it in 75 randomly selected real changed lines from the four software systems. The chosen threshold achieves a precision of 73.2%. Although we consider the precision to be enough for this study, future research needs to be conducted to determine line changes in source code diffs.
4. Our tool detects changes of 13 Mockito APIs. These 13 APIs happen to be in Mockito since its very first version and past literature (Mostafa and Wang 2014) shows that they are the most used APIs. Choosing these 13 APIs did not allow us to investigate "adoption patterns" (i.e., how developers take advantage of a newly introduced mocking API). Indeed, Mockito development history shows that new APIs are constantly being added, and thus, we leave as future work to explore their adoption.
5. As explained in Section 3, the first two authors manually classified the types of technical dept. Since this process was done simultaneously (the authors were seated in the same room next to each other) and they were discussing each technical dept, no validation of agreement was needed.
6. Our tool also is able to statically detect changes in lines that involve Mockito API, *e.g.* `verify(mock).action()`. In practice, different implementation strategies may result in different results. As an example, if a test class A contains one `verify` line in each test (thus, several `verifys` in the source code), and another test class B encapsulates this call in a private method (thus, just a single `verify` call in the source code), the change analysis in both classes will yield different results. However, as our analysis is performed in scale (*i.e.* we analyzed *all* the commits in the main branch of the four systems), we conjecture that such small differences do not have a large impact on the implications of our study.
7. In Section 5.2, we investigated the relation between mocks and code quality. In the comparison between mocked and not mocked classes, we controlled for the fact that classes may be tested a different number of times. To this aim, we divided the classes into four categories (Low, Medium, High, Very High), according to the number of times they were tested. To choose the threshold of the categories, we used the 80th, 90th and 95th percentiles. Even though these percentiles have been already used in previous research (Alves et al. 2010; Aniche et al. 2016), different thresholds may lead to different results. It is in our future agenda to better investigate the relation between mocks and code quality, with a deeper analysis on the characteristics of the most mocked classes.

Internal validity Threats to *internal validity* concern factors we did not consider that could affect the variables and the relations being investigated:

1. We performed manual analysis and interviews to understand why certain dependencies are mocked and not mocked. A single developer does not know all the implementation decisions in a software system and may think and behave differently from the rest of the team. Hence, developers may wrongly choose to mock/not mock a class. We tried to mitigate this issue in several ways: (1) during interviews, by explicitly discussing both their point of view as well as the "rules" that are followed by the entire team, (2) and by presenting the results of RQ_1 and asking them to help us interpret it, with the hope that this would help them to see the big picture of their own project; (3) regarding the manual analysis, by analyzing large and important OSS projects with stringent policies on source code quality, and (4) by quantitatively analyzing a large set of production classes (a total of 38,313 classes) and their mocking decisions.
2. During the interview, 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 (Nederhof 1985)) 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. To perform the manual analysis in RQ_5, we randomly selected 100 changes of each system (which gives a CL=95%, CI=10). During the analysis, changes in larger commits may appear more often than changes in smaller commits, *e.g.* several changes in mock objects may be related to the same large refactoring commit. We observed such effect particularly during the analysis of VRaptor. Due to the amount of changes we analyzed in the four systems, we do not expect significant variation in the results. Nevertheless, it is part of our future work to perform stratified sampling and compare the results.

External validity Threats to *external validity* concern the generalization of results:

1. 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.
2. Similarly, we are not able to generalize the results we found regarding the evolution of the mocks. As a way to reduce the threat, the four analyzed systems present different characteristics and focus on different domains. Future replication should be conducted to consolidate our results.
3. 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.

6 Related Work

In this section, we present related work on: (1) empirical studies on the usage of mock objects, (2) studies on test evolution and test code smells (and the lack of mocking in such studies), (3) how automated test generation tools are using mock objects to isolate external

dependencies, and (4) the usage of pragmatic unit testing and mocks by developers and their experiences.

Empirical studies on the usage of mock objects Despite the widespread usage of mocks, very few studies analyzed current mocking practices. Mostafa and Wang (2014) 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, Mockito being the most widely used (70%). In addition, software testers seem to mock only a small portion of all dependency classes of a test class. On average, about 17% of dependency classes are mocked by the software testers. This is also observed in the number of mock objects in test classes: 45% of test files contain just a single mock, and 21% contain just two mocks; only 14% contain five or more mock objects. Their results also show that about 39% of mocked classes are library classes. This implies that software testers tend to mock more classes that belong to their own source code, when compared with library classes. In terms of API usage, `Mockito.verify()` (used to perform assertions in the mock object), `Mockito.mock()` (used to instantiate a mock), and `Mockito.when()` (used to define the behavior of the mock) are by far the most used methods. This is in line with the results for our RQ_5. They also observed similar results for EasyMock (the second most popular mock framework in their study).

Marri et al. (2009) investigated the benefits as well as challenges of using mock objects to test file-system-dependent software. Their 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) allow the generation of high-covering unit tests. However, according to the authors, mock objects can cause problems when the code under test involves interactions with multiple APIs that use the same data or interact with the same environment.

Test evolution and code smells (and the lack of mocking) Studies that focus on the evolution of test code, test code smells, and test code bugs have been conducted. However, they currently do not take mocks as a perspective, which should be seen as suggestions for future work. Vahabzadeh et al. (2015) mined 5,556 test-related bug reports from 211 projects from the Apache Software Foundation to understand bugs in test code. Results show that false alarms are mostly caused by semantic bugs (25%), flaky tests (21%) environment (18%), and resource handling (14%). Among the environmental alarms, 61% are due to platform-specific failures, caused by operating system differences. Authors did not report any bugs related to mock objects, which leaves us to conclude that either these tests did not make use of mock objects, or mocks were not taken into account during their analysis.

Zaidman et al. (2008) investigated how test and production code co-evolve in both open source and industrial projects. Authors found that production code and test code are usually modified together, that there is no clear evidence of a testing phase preceding a release, and that only one project in their studied sample used Test-Driven Development (they approximated it by looking to tests and production files committed together). In another study, Vonken and Zaidman (2012) performed a two-group controlled experiment involving 42 participants with the focus on investigating whether having unit tests available during refactoring leads to quicker refactorings and higher code quality. Results, however, indicate that having unit tests available during refactoring does not lead to quicker refactoring or higher-quality code after refactoring. Although the system used in the experiment made use of

mocks, authors did not use mocks as control, and thus, the paper does not shed light on the relationship between mock objects and test refactoring.

van Deursen et al. (2001) coined the term *test smells* and defined the first catalog of eleven poor design solutions to write tests, together with refactoring operations aimed at removing them. Such a catalog has been then extended more recently by practitioners, such as Meszaros who defined 18 new test smells (Meszaros 2007). Although the catalog contains the `Slow tests` smell, which a solution could be the use of mock objects, there are no smells specific to the usage of mocks.

Such investigation can be important to the community, as it is known that test smells happen in real systems and have a negative impact on their maintenance. Greiler et al. 2013, 2013 showed that test smells affecting test fixtures frequently occur in industry. Motivated by this prominence, Greiler et al. presented TESTHOUND, a tool able to identify fixture-related test smells such as *General Fixture* or *Vague Header Setup* (Greiler et al. 2013). Van Rompaey et al. (2007) also proposes detection strategies for *General Fixture* and *Eager Test*, although their empirical study shows that the common often misses smelly instances.

Bavota et al. (2015) studied the diffusion of test smells in 18 software projects and their effects on software maintenance. As a result, authors found that 82% of test classes are affected by at least one test smell. Interestingly, the same problem happens in test cases that are automatically generated by testing tools (Palomba et al. 2016). In addition, Bavota et al. (2015) show that the presence of test smells has a strong negative impact on the comprehensibility of the affected classes. Tufano et al. (2016) also showed that test smells are usually introduced during the first commit involving the affected test classes, and in almost 80% of the cases they are never removed, essentially because of poor awareness of developers.

Automatic test generation and mocks Taneja et al. (2010) 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 these reasons, authors proposed an "Automated Test Generation" for Database Applications using mock objects, demonstrating that with this technique they could achieve better test coverage.

Arcuri et al. (2014) applied bytecode instrumentation to automatically separate code from any external dependency. After implementing a prototype in EvoSuite (Fraser and Arcuri 2011) that was able to handle environmental interactions such as keyboard inputs, file system, and several non-deterministic functions of Java, authors show that EvoSuite was able to significantly improve the code coverage of 100 Java projects; in some cases, the improvement was in the order of 80% to 90%.

Another study, also by Arcuri et al. (2017), focused on extending the EvoSuite unit test generation tool with the ability to directly access private APIs (via reflection) and to create mock objects using Mockito. Their experiments on the SF110 and Defects4J benchmarks confirm the anticipated improvements in terms of code coverage and bug finding, but also confirm the existence of false positives (due to the tests that make use of reflection, and thus, depend on specificities of the production class, *e.g.* a test accessing a private field will fail if that field is later renamed).

Finally, Li et al. (2016) proposed a technique that combines static analysis, natural language processing, backward slicing, and code summarization techniques to automatically generate natural language documentation of unit test cases. After evaluating the tool with a set of developers, authors found out that the descriptions generated by their tool are easy to read and understand. Interestingly, a developer said: *"mock-style tests are not well*

described.", suggesting that the tool may need improvement in tests that make use of mock objects.

Pragmatic unit testing and mock Several industry key leaders and developers affirm that the usage of mock objects can bring benefits to testing. Such experience reports call for in-depth, scientific studies on the effects of mocking.

Mackinnon et al. (2001), for example, in their chapter on a book about Extreme Programming, 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.

Karlesky et al. (2007), after their real-world experience in testing embedded systems, present a holistic set of practices, platform independent tools, and a new design pattern (Model Conductor Hardware - MCH) that together produce: good design from tests programmed first, logic decoupled from hardware, and systems testable under automation. Interestingly, the authors show how to mock hardware behavior to write unit tests for embedded systems.

Similarly, Kim (2016) 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.

7 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. Studying the evolution of mocks, we found that they are generally introduced at the inception of test classes and tend to stay within these classes for the entire lifetime of the classes. Mocks changes in an equally frequent way for changes to the production code that they simulate and for changes to the test code (*e.g.* refactoring) that use them.

Our future agenda includes understanding the relationship between code quality metrics and the use of mocking practices, investigating the reasons behind mock deletions, and analyzing adoption patterns as well as the differences of mock adoptions in dynamic languages.

Acknowledgment This project has received funding from the European Union's H2020 programme under the Marie Sklodowska-Curie grant agreement No 642954. A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

References

Alves TL, Ypma C, Visser J (2010) Deriving metric thresholds from benchmark data. In: IEEE international conference on software maintenance ICSM. https://doi.org/10.1109/ICSM.2010.5609747

Aniche M, Treude C, Zaidman A, Deursen AV, Gerosa MA (2016) SATT: tailoring code metric thresholds for different software architectures. In: Proceedings - 2016 IEEE 16th international working conference on source code analysis and manipulation, SCAM 2016, pp 41–50. https://doi.org/10.1109/SCAM.2016.19

Arcuri A, Fraser G, Galeotti JP (2014) Automated unit test generation for classes with environment dependencies. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering, pp 79–90. ACM

Arcuri A, Fraser G, Just R (2017) Private api access and functional mocking in automated unit test generation. In: 2017 IEEE international conference on software testing, verification and validation (ICST), pp 126–137. IEEE

Baeza-Yates R, Ribeiro-Neto B et al (1999) Modern information retrieval, vol 463. ACM Press, New York

Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2015) Are test smells really harmful? An empirical study. Empir Softw Eng 20(4):1052–1094. https://doi.org/10.1007/s10664-014-9313-0. http://link.springer.com/10.1007/s10664-014-9313-0

Beck K (2003) Test-driven development: by example. Addison-Wesley Professional, Boston

Biegel B, Soetens QD, Hornig W, Diehl S, Demeyer S (2011) Comparison of similarity metrics for refactoring detection. In: Proceedings of the 8th working conference on mining software repositories, pp 53–62. ACM

Bruntink M, Van Deursen A (2004) Predicting class testability using object-oriented metrics. In: 4th IEEE international workshop on source code analysis and manipulation, 2004. pp 136–145. IEEE

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: 2010 7th IEEE working conference on mining software repositories (MSR), p 31–41. IEEE

EasyMock (2016) http://easymock.org. Online, Accessed 3 Feb 2016

Evans E (2004) Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, Boston

Fowler M (2002) Patterns of enterprise application architecture. Addison-wesley Longman Publishing Co. Inc, Boston

Fowler M, Beck K, Hansson DH (2014) Is tdd dead? https://plus.google.com/events/ci2g23mk0lh9too9bgbp3rbut0k. Last access in July, 2018

Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, pp 416–419. ACM

Freeman S, Mackinnon T, Pryce N, Walnes J (2004) Mock roles, objects. In: Companion to the 19th annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications, pp 236–246. ACM

Freeman S, Pryce N (2009) Growing object-oriented software, guided by tests. Pearson Education, London

Greiler M, van Deursen A, Storey MA (2013) Automated detection of test fixture strategies and smells. In: 2013 IEEE 6th international conference on software testing, verification and validation, pp 322–331. https://doi.org/10.1109/ICST.2013.45

Greiler M, Zaidman A, van Deursen A, Storey MA (2013) Strategies for avoiding text fixture smells during software evolution. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), pp 387–396. IEEE

Hamill P (2004) Unit test frameworks: tools for high-quality software development. O'Reilly Media

Hanington B, Martin B (2012) Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions. Rockport Publishers, Beverly

Henderson F (2017) Software Engineering at Google. arXiv:1702.01715

Hess MR, Kromrey JD (2004) 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. https://doi.org/10.1088/1751-8113/44/8/085201. http://www.coedu.

usf.edu/main/departments/me/documents/cohen.pdf arXiv:1011.1669 http://stacks.iop.org/1751-8121/44/i=8/a=085201?key=crossref.abc74c979a75846b3de48a5587bf708f
Hunt A, Thomas D (2004) Pragmatic unit testing in c# with nunit. The Pragmatic Programmers
JMock (2016) http://www.jmock.org. Online, Accessed 3 Feb 2016
Kaczanowski T (2012) Practical Unit Testing with testNG and Mockito. Tomasz Kaczanowski
Karlesky M, Williams G, Bereza W, Fletcher M (2007) 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
Kim SS (2016) Mocking embedded hardware for software validation. Ph.D thesis
Kotlin (2016) https://kotlinlang.org. Online; Accessed 3 Feb 2016
Langr J, Hunt A, Thomas D (2015) Pragmatic unit testing in java 8 with JUnit. Pragmatic Bookshelf
Li B, Vendome C, Linares-Vásquez M, Poshyvanyk D, Kraft NA (2016) Automatically documenting unit test cases. In: 2016 IEEE international conference on software testing, verification and validation (ICST), pp 341–352. IEEE
Mackinnon T, Freeman S, Craig P (2001) Extreme Programming Examined. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN:0-201-71040-4
Marri MR, Xie T, Tillmann N, De Halleux J, Schulte W (2009) An empirical study of testing file-system-dependent software with mock objects. AST 9:149–153
McCabe T (1976) A complexity measure. IEEE Trans Softw Eng SE-2(4):308–320. https://doi.org/10.1109/TSE.1976.233837. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702388. http://ieeexplore.ieee.org/document/1702388/
Meszaros G (2007) xUnit test patterns: Refactoring test code. Pearson Education, London
Mostafa S, Wang X (2014) An empirical study on the usage of mocking frameworks in software testing. In: 2014 14th international conference on quality software, pp 127–132. IEEE. https://doi.org/10.1109/QSIC.2014.19. http://ieeexplore.ieee.org/document/6958396/
Mock (2016) https://github.com/testing-cabal/mock. Online, Accessed 3 Feb 2016
Mocker (2016) https://labix.org/mocker. Online, Accessed 3 Feb 2016
Mockito (2016) http://site.mockito.org. Online, Accessed 3 Feb 2016
MyBatis (2016) http://www.mybatis.org/. Online, Accessed 3 Feb 2016
Nederhof AJ (1985) Methods of coping with social desirability bias: a review. Eur J Social Psychol 15(3):263–280
Osherove R (2009) The art of unit testing: with examples in.NET Manning
Palomba F, Di Nucci D, Panichella A, Oliveto R, De Lucia A (2016) On the diffusion of test smells in automatically generated test code: an empirical study. In: Proceedings of the 9th international workshop on search-based software testing, pp 5–14. ACM
Pereira F (2014) Mockists are dead. long live classicists. https://www.thoughtworks.com/insights/blog/mockists-are-dead-long-live-classicists. Last access in July, 2018
PowerMock (2016) https://github.com/powermock/powermock. Online, Accessed 3 Feb 2016
Razali NM, Wah YB (2011) Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. J Statist Model Anal 2(1):21–33. https://doi.org/10.1515/bile-2015-0008
Rugg G (2005) Article picture sorts and item sorts. Computing 22(3):94
Runeson P (2006) A survey of unit testing practices. IEEE Softw 23(4):22–29. https://doi.org/10.1109/MS.2006.91. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1657935
Samimi H, Hicks R, Fogel A, Millstein T (2013) Declarative mocking categories and subject descriptors, pp 246–256
Spadini D (2017) To Mock or Not To Mock? Online Appendix. https://doi.org/10.4121/uuid:fce8653c-344c-4dcb-97ab-c9c1407ad2f0
Spadini D, Aniche M, Bacchelli A, Bruntink M (2017) MockExtractor. The tool is available at http://www.doi.org/10.5281/zenodo.1475900
Spadini D, Aniche M, Bruntink M, Bacchelli A (2017) To mock or not to mock?: an empirical study on mocking practices. In: Proceedings of the 14th international conference on mining software repositories, pp 402–412. IEEE Press
Spencer D (2004) Card sorting: a definitive guide. http://boxesandarrows.com/card-sorting-a-definitive-guide/
Taneja K, Zhang Y, Xie T (2010) MODA: automated test generation for database applications via mock objects. In: Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10, p 289. ACM Press, New York. https://doi.org/10.1145/1858996.1859053. http://portal.acm.org/citation.cfm?doid=1858996.1859053
Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016, pp 4-15. ACM, New York

Vahabzadeh A, Fard AM, Mesbah A (2015) An empirical study of bugs in test code. In: 2015 IEEE international conference on software maintenance and evolution (ICSME), pp 101–110. IEEE
van Deursen A, Moonen L, Bergh A, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP), pp 92–95
Van Rompaey B, Du Bois B, Demeyer S, Rieger M (2007) On the detection of test smells: a metrics-based approach for general fixture and eager test. IEEE Trans Softw Eng 33(12):800–817. https://doi.org/10.1109/TSE.2007.70745
Vonken F, Zaidman A (2012) Refactoring with unit testing: a match made in heaven? In: 2012 19th working conference on reverse engineering (WCRE), pp 29–38. IEEE
Weyuker E (1998) Testing component-based software: a cautionary tale. IEEE Softw 15(5):54–59. https://doi.org/10.1109/52.714817. http://ieeexplore.ieee.org/document/714817/
Wilcoxon F (1946) Individual comparisons of grouped data by ranking methods. J Econ Entomol 39(6):269. https://doi.org/10.2307/3001968
Wynne M, Hellesoy A (2012) The cucumber book: behaviour-driven development for testers and developers. Pragmatic Bookshelf
Wynne M, Hellesoy A (2012) The cucumber book: Behaviour-driven development for testers and developers. Pragmatic Bookshelf
Xie T, Tillmann N, De Halleux J, Schulte W (2010) Future of developer testing: Building quality in code. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, pp 415–420. ACM
Zaidman A, Rompaey BV, Demeyer S, Deursen AV (2008) Mining software repositories to study co-evolution of production and test code. In: 2008 1st international conference on software testing, verification, and validation, vol 3, pp 220–229. https://doi.org/10.1109/ICST.2008.47
Zaidman A, Van Rompaey B, Demeyer S, Van Deursen A (2008) Mining software repositories to study co-evolution of production & test code. In: 2008 1St international conference on software testing, verification, and validation, pp 220–229. IEEE

Davide Spadini is a PhD student at the University of Technology of Delft and he is currently working as a researcher at SIG in Amsterdam. He received his B.Sc. and M.Sc. in Computer Science from the University of Verona and Trento, Italy. Since he started the PhD in 2016, he focused his research activity in the software testing area. This includes (but not limited to) software testing practices, test code quality, test code review, and test code maintenance.

Maurício Aniche is an Assistant Professor at Delft University of Technology, The Netherlands. Maurício helps developers to effectively maintain, test, and evolve their software systems. His current research interests are systems monitoring and DevOps, empirical software engineering, and software testing.

Magiel Bruntink is Head of Research of the Software Improvement Group, a consultancy that focuses on the automatic analysis of software quality and related decision making. He has a scientific background in program analysis and empirical study. Most of his work consists of mixed industry-academic projects, positioned within the software engineering domain.

Alberto Bacchelli is an SNSF Professor in Empirical Software Engineering in the Department of Informatics in the Faculty of Business, Economics and Informatics at the University of Zurich, Switzerland. He received his B.Sc. and M.Sc. in Computer Science from the University of Bologna, Italy, and the Ph.D. in Software Engineering from the Università della Svizzera Italiana, Switzerland. Before joining the University of Zurich, he has been assistant professor at Delft University of Technology, The Netherlands where he was also granted tenure. His research interests include peer code review, empirical studies, and the fundamentals of software analytics.

Affiliations

Davide Spadini[1,2] · Maurício Aniche[1] · Magiel Bruntink[2] · Alberto Bacchelli[3]

Maurício Aniche
M.F.Aniche@tudelft.nl

Magiel Bruntink
m.bruntink@sig.eu

Alberto Bacchelli
bacchelli@ifi.uzh.ch

1 Delft University of Technology, Delft, Netherlands

2 Software Improvement Group, Amsterdam, Netherlands

3 University of Zurich, Zürich, Switzerland