Assessing Mock Classes: An Empirical Study

Gustavo Pereira, Andre Hora
ASERG Group, Department of Computer Science (DCC)
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
{*ghapereira, andrehora*}@*dcc.ufmg.br*

***Abstract*—During testing activities, developers frequently rely on dependencies (*e.g.,* web services, etc) that make the test harder to be implemented. In this scenario, they can use mock objects to emulate the dependencies' behavior, which contributes to make the test fast and isolated. In practice, the emulated dependency can be dynamically created with the support of mocking frameworks or manually hand-coded in mock classes. While the former is well-explored by the research literature, the latter has not yet been studied. Assessing mock classes would provide the basis to better understand how those mocks are created and consumed by developers and to detect novel practices and challenges. In this paper, we provide the first empirical study to assess mock classes. We analyze 12 popular software projects, detect 604 mock classes, and assess their content, design, and usage. We find that mock classes: often emulate domain objects, external dependencies, and web services; are typically part of a hierarchy; are mostly public, but 1/3 are private; and are largely consumed by client projects, particularly to support web testing. Finally, based on our results, we provide implications and insights to researchers and practitioners working with mock classes.**

***Keywords*-Software Testing; Mocks; Test Double; Software Maintenance; Mining Software Repositories**

I. Introduction

Software testing is a key practice in modern software development. Often, during testing activities, developers are faced with dependencies (*e.g.,* web services, database, etc) that make the test harder to be implemented [1]. In this scenario, developers can either instantiate these dependencies inside the test or use mock objects to emulate the dependencies' behavior [2], [3]. The use of mock objects can contribute to make the test fast, isolated, repeatable, and deterministic [1]. A test case that for example relies on an unstable and slow external service can mock this dependency to be stable and faster. To support the learning of mock objects, technical literature is available for distinct programming languages and ecosystems (*e.g.,* [1], [4]–[7]).

In practice, there are two solutions to adopt mock objects. The emulated dependency can be dynamically created with the support of mocking frameworks or manually hand-coded in *mock classes* [1]. Indeed, mocking frameworks are quite popular nowadays and are found in distinct software ecosystems. For example, JavaScript developers may use SinonJS[1] and Jest,[2] which is supported by Facebook; Java developers can rely on Mockito[3] while Python provides unittest.mock[4] in its core library. The other solution to create mock classes is by hand, that is, manually creating emulated dependencies so they can be used in test cases. In this case, developers do not need to rely on any particular mocking framework since they can directly consume the mock class. For example, to facilitate web testing, the Spring web framework includes a number of classes dedicated to mocking.[5] Similarly, the Apache Camel integration framework provides mocking classes to support distributed and asynchronous testing.[6] That is, in those cases, instead of using a mocking framework to simulate a particular dependency, developers can directly use mocking classes on their test cases, such as `MockServer`, `MockHttpConnection`, `MockServlet`, etc.

Past research showed that mocking frameworks are largely adopted by software projects [8] and that they may indeed support the creation of unit tests [2], [3], [9], [10]. Moreover, recent research showed how and why practitioners use mocks and the challenges faced by developers [2], [3]. However, those researches are restricted to the context of mocking frameworks. To the best of our knowledge, mock classes have not yet been studied by the research literature. In this context, some important questions are still open, such as: what dependencies are emulated by those mock classes? are they any different from framework mocks? how are mock classes designed and used by developers? Thus, assessing mock classes would provide the basis (i) to understand how those mock objects are created and consumed by developers and (ii) to detect novel practices and challenges.

In this paper, we provide the first empirical study to assess mock classes. We analyze 12 popular software projects and detect 604 mock classes. Thus, we propose the following research questions to assess their content, design, and usage:

- *RQ1 (Content): What is the content of mock classes?*

[1]https://sinonjs.org
[2]https://jestjs.io
[3]https://site.mockito.org
[4]https://docs.python.org/3/library/unittest.mock.html
[5]https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html#mock-objects
[6]https://camel.apache.org/components/latest/mock-component.html

We manually categorize the 604 mock classes with respect to the dependency they are emulating. We observe that mock classes typically emulate domain objects, external dependencies, and web services. The most and least frequent categories are essentially the same that developers create when relying on mocking frameworks.

- *RQ2 (Design): How are mock classes designed?* We assess the structural details of the mock classes. We find that mock classes often extend other classes or implement interfaces; around 70% are public and can be reused, while 30% have restrictive visibility; and mock and regular classes have the same number of methods.
- *RQ3 (Usage): How are mock classes used by developers?* In this analysis, we focus on the client-side. With the support of the Boa platform [11], we assess millions of client projects and detect that mock classes are largely consumed, particularly to emulate web services. However, the usage is very concentrated on a few classes: 10 classes are consumed by 76% of the clients.

Based on our results, we provide insights and practical implications to researchers and practitioners by discussing topics as (i) the novel empirical data on mock classes, (ii) the reusability and lack of visibility of the mock classes, and (iii) the widespread usage of the mock classes. We reveal novel quantitative and qualitative empirical data about the creation of mock classes, which can guide practitioners in charge of maintaining them. We shed light on the over creation of private mock classes, which can be harmful to the overall project maintainability. We present that the usage of mock classes is not restricted to the target projects of this study, but it seems to be widespread, thus, they should be maintained with care because client projects can be impacted.

Contributions. The contributions of this research are threefold: (i) we provide the first empirical study on mock classes, from both quantitative and qualitative perspectives; (ii) we perform a large analysis of mock classes to better understand their content, design, and usage; and (iii) we propose insights and practical implications to researchers and practitioners working on and consuming mock classes.

Organization. Section II presents the background, describing mocks, test doubles, and mock classes. Section III details the study design, while Section IV the results. Section V discusses the results, providing insights and implications. Section VI details the threats to validity. Finally, Section VII presents the related work and Section VIII concludes.

II. Background

A. Mocks and Test Doubles

According to Meszaros [1], mock objects "*replace an object the system under test (SUT) depends on with a test-specific object that verifies it is being used correctly by the SUT [...]. Mock Object is a powerful way to implement behavior verification.*" As originally proposed, during the test, mock objects are configured with the values with which they should respond to the SUT and the method calls that are expected to be made from the SUT [1]. A mock object is a particular type of test double [1], which has a lighter definition: "*it replaces a component on which the SUT depends with a test-specific equivalent*". In addition to mock objects, other test doubles are stubs, spies, and fake objects, each one with its nuances [1] (*e.g.,* while mocks focus on behavior verification, other test doubles focus on state verification). In short, test double can be defined as "*the broadest term available to describe any fake thing introduced in place of a real thing for the purpose of writing an automated test*".[7]

Despite the formal definitions, the state of the practice is to frequently use the terms mock objects and test doubles interchangeably, for example:

- Robert Martin (author of Clean Code [12]): "*The word "mock" is sometimes used in an informal way to refer to the whole family of objects that are used in tests.*"[8]
- Martin Fowler (author of Refactoring [13]): "*The term Mock Objects has become a popular one to describe special case objects that mimic real objects for testing.*"[9]
- Harry Percival (author of TDD with Python [7]): "*I'm using the generic term "mock", but testing enthusiasts like to distinguish other types of a general class of test tools called Test Doubles [...] The differences don't really matter for this book.*"[10]
- testdouble.js wiki about testing: "*There are several subtypes of test doubles, but most tools do a poor job either following a conventional nomenclature [...].*"[11]

The terminology around the types of test doubles is confusing and inconsistent, hence different people use distinct terms to mean the same thing [1]. As a result, this leads to an endless discussion[12] on such theme [14]. In the following subsection, we present how we assess mock objects in the face of such inconsistency.

B. Mock Classes

As briefly described in the introduction, mock objects can be either (1) dynamically created with the support of mocking frameworks or (2) manually hand-coded in mock classes [1]. For example, suppose a developer wants to simulate a dependency on `HttpRequest` when performing web testing. In this case, he could either use a mocking framework (such as Mockito, Jest, SinonJS, etc) to dynamically create a mock object that simulates

[7]https://github.com/testdouble/contributing-tests/wiki/Test-Double
[8]https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html
[9]https://martinfowler.com/articles/mocksArentStubs.html
[10]http://www.obeythetestinggoat.com/book/chapter_mocking.html
[11]https://github.com/testdouble/contributing-tests/wiki/Test-Double
[12]Example: https://bit.ly/3d1XumC

`HttpRequest` or he could create a mock class by hand (*e.g.,* `MockHttpRequest`) to simulate `HttpRequest`. Interestingly, writing the mock classes forces developers to give those mocks names, so they can be reused in other tests; moreover, mock classes should be somehow designed to be consumed by clients [15].

To avoid the inconsistency around mock objects and test doubles that was stated in the previous subsection, in this study we rely on a lightweight solution to detect mock classes. We consider mock classes the ones that are explicitly declared as mocks, that is, the classes that are named with the term "mock". For example, the class `MockServer-HttpRequest`[13] provided by the Spring framework is a mock implementation of the real `ServerHttpRequest`. Thus, due to the lack of consistency in the terminology, we recognize that we may detect other test doubles and not only mock objects. Consequently, in this study, the usage of the term mock resembles the widespread usage in the larger field of software development, that is, mocking being adopted as a generic term to represent any test double.

III. Study Design

A. Selecting the Software Projects

In this study, we aim to assess mock classes provided by real world and relevant software projects. We then select the 10 most popular Java systems hosted on GitHub based on the star metric [16], [17], which is a proxy for popularity. In addition, two other important projects were included: Apache Lucene-Solr and Apache Camel. The 12 selected projects are presented in Table I. The most popular project of the sample is Elasticsearch (48.3k stars), while the least popular one in it is Camel (3.2k stars). The selected projects cover distinct software domains: web framework (Spring Boot and Spring Framework), search engine (Elasticsearch and Lucene-Solr), asynchronous/event library (RxJava and EventBus), HTTP client (OkHttp and Retrofit), Java support library (Guava), RPC framework (Dubbo), integration framework (Camel), and analytics engine (Spark).

B. Detecting Mock Classes

The next step is to detect mock classes.[14] First, for each project, we extract all classes in the Java files, including nested ones. This result is presented in the column "Classes" in Table I. The largest project in number of classes is Camel (18,757 classes), while the smallest is EventBus (163 classes); in total, those projects include over 78.8k classes. We then perform two analyses to assess (1) mocks created with the support of frameworks and (2) mock classes.

1. Mocks created with the support of mocking frameworks. Before presenting the mock classes, for comparison purposes, we first present the mocks created with the support of mocking frameworks. We verify the number of classes that are mocked using the most popular mocking framework in Java, Mockito (column "Classes Mocked Using Mockito" in Table I).[15] To this aim, we verify the classes being mocked with the Mockito API `Mockito.mock()`, as done by the related literature [2], [3]. For instance, Elasticsearch mocks 524 classes via Mockito, while Spring Boot mocks 367; in total, we find 1,519 classes being mocked with this framework. Only three systems (OkHttp, Retrofit, and EventBus) do not create any mock with Mockito. This shows that the usage of mocking is widespread to support testing.

Table I
Selected projects and detected mock classes.

Project	Stars	Classes	Classes Mocked Using Mockito	Mock Classes
Elasticsearch	48.3k	16,097	524	138
Spring Boot	47.1k	7,845	367	33
RxJava	42.4k	2,766	21	0
Guava	36.8k	6,174	19	12
OkHttp	36.8k	210	0	1
Spring Fw.	36.5k	10,980	272	127
Retrofit	35.5k	303	0	4
Dubbo	32k	2,104	60	58
Spark	25.8k	1,850	23	0
EventBus	22.4k	163	0	0
Lucene-Solr	3.4k	11,583	35	132
Camel	3.2k	18,757	198	99
Total	-	78,832	1,519	604

2. Mock classes. Lastly, in the column "Mock Classes" in Table I, we present the target classes of this study, that is, the mock classes. We consider that a class is a mock when:

- It is explicitly declared as a mock class, that is, its name includes the term "mock", but not "mockito" *and*
- It does not use the most popular mocking framework in Java (*i.e.,* it does not import any Mockito class) *and*
- It is not a test class, that is, its name does not end with the suffix "Test" nor "Tests" (which is the guideline to create testing classes in Java [2], [3]).

Following this approach, we detect 604 mock classes. Six projects have over 30 mock classes: Elasticsearch (138), Camel (99), Spring Framework (127), Lucene-Solr (132), Dubbo (58), and Spring Boot (33). Three projects have 1, 4, and 12 mock classes, while three projects have none.

Table II presents some examples of mock classes. As we can observe, the class names are very distinct and the dependencies they are simulating are diverse. For example, based on their names, they are likely to be related to web services (*e.g.,* `MockWebSession` and `MockMvc`), network services (*e.g.,* `MockSocketChannel` and `MockMockTelnet-Handler`), external dependencies (*e.g.,* `MockGitHub` and `AmazonECSClientMock`), to name a few.

[13]https://bit.ly/3bQF9ZH
[14]The collected data is available online: https://github.com/ghapereira/assessing_mock_classes_data
[15]We only verified Mockito because it is the *de facto* mocking framework in Java. For comparison, Mockito has over 10,000 stars on GitHub, while EasyMock has 650 stars and JMock has only 119.

Table III presents the most frequent terms on the names of the 604 mock classes (after removing "Mock"). We notice the presence of some generic terms such as Client, Filter, and Service, but also some specific ones as Amazon, Http, and Mvc. The top terms are Client (55 occurrences), Filter (36), Amazon (33), Factory (33), and Request (33).

Table II
EXAMPLES OF MOCK CLASSES.

Project	**Examples**
Elasticsearch	MockSocketChannel, MockBlobStore, MockMessage
Spring Boot	MockCachingProvider, MockFilter, MockServlet
Guava	MockCallback, MockRunnable, MockExecutor
Spring Fw.	MockConnection, MockWebSession, MockMvc
Retrofit	MockGitHub, MockRetrofit, MockRetrofitIOException
Dubbo	MockTelnetHandler, MockDirectory, MockThreadPool
Lucene-Solr	MockTrigger, MockScorable, MockTimerSupplier
Camel	AmazonECSClientMock, MockRest, MockEndpoint

Table III
MOCK-RELATED TERMS.

Pos	**Term**	**#**	**Pos**	**Term**	**#**
1	Client	55	6	Http	31
2	Filter	36	7	Script	27
3	Amazon	33	8	Mvc	27
4	Factory	33	9	Server	25
5	Request	33	10	Service	25

C. Assessing the Research Questions

1) RQ1 (Content): Our first research question is intended to assess the content of the mock classes, that is, what type of dependency they are simulating. We rely on the categories proposed by a previous related study in the context of mocking frameworks [2], [3]. This has two advantages: (1) we keep consistency with previous research studies and (2) we can directly compare our results with previous ones. Thus, we adopt the categories proposed by Spadini *et al.* [2], [3]: domain object, database, native Java libraries, web service, external dependency, and test support. Besides, after inspecting our dataset, a new category is considered: network service. Those categories are defined as follow:

- *Domain object:* classes that are mock to business rules of the system.
- *Database:* classes that are mock to SQL/NoSQL database library.
- *Native Java libraries:* classes that are mock to the Java native libraries.
- *Web service:* classes that are mock to some web action.
- *External dependency:* classes that are mock to external libraries.
- *Test support:* classes that support testing itself.
- *Network service:* classes that support network services.

To categorize the mock classes, we perform a qualitative analysis based on (1) class names, (2) documentation analysis, and (3) source code inspection. Some mock classes can be resolved based on the analysis of their names. For example, `MockWebServer`, `MockMvc`, and `MockRest` are examples of the category web service, while `MockSocketChannel`, `MockTcpChannelFactory`, `MockTcpReadWriteHandler` are examples of network service. Other classes, such as `MockTokenizer`, `MockTargetSource`, and `MockAnalyzer` are not straightforward and require either documentation analysis or code inspection to detect their categories.

2) RQ2 (Design): In this second research question, we aim to study how mock classes are designed. We then assess some important structural aspects, as follows:

- Inheritance and Interface: mock classes can be created by extending classes or implementing interfaces. We assess whether the mock classes are more likely to implement interfaces, extend classes, or none of them. Our final goal is to better understand whether mock classes are part of hierarchies or are standalone classes.
- Visibility: mock classes can have a public scope to be used all over the project (or by external clients) or they can have private scope to be only locally used. Thus, we explore the visibility of the mock classes to verify whether they are likely to be reusable or not.
- Number of methods: we analyze the number of methods provided by the mock classes. We also compare with the number of methods provided by regular classes, which are randomly selected following the same distribution of mock classes per system. We aim to understand whether mock classes are likely to be larger (which may demand more effort to create and maintain) or smaller (which may demand less effort).

3) RQ3 (Usage): In this research question, we focus on the client-side. We rely on the ultra-large-scale dataset Boa [11], which includes over 2 million Java systems, to detect whether mock classes are used in the wild. Specifically, we perform a query on all Java systems looking for import statements with the term "mock". We then filter out classes with the terms "Mockito", "Mockery", and "EasyMock" (*i.e.,* classes related to mocking frameworks), and also the ones that end with "Test" or "Tests" (*i.e.,* testing classes in Java). This way, we find 6,444 distinct classes that are imported 147,433 times. Table IV summarizes the most frequent terms in these mock classes. As we can notice, the most frequent terms are Service, Factory, and Context. Lastly, to answer our research question, we assess the frequency of our 604 mock classes in this dataset as well as the most recurrent categories.

Table IV
MOCK-RELATED TERMS IN BOA.

Pos	Term	#	Pos	Term	#
1	Service	411	6	Provider	161
2	Factory	299	7	Impl	148
3	Context	224	8	Request	145
4	Manager	202	9	Modules	138
5	Data	167	10	Test	136

IV. RESULTS

A. RQ1 (Content): What is the content of mock classes?

Table V presents the categories of the mock classes detected after our manual classification. As we notice, the most common category is domain object (35%), followed by external dependency (23%) and web service (15%). Domain object (*i.e.,* classes that are mock to business rules) is present in 211 classes; examples mock classes in this category include: `MockAction` (Elasticsearch) and `MockCoreDescriptor` (Lucene-Solr). The second most frequent is external dependency (138 mock classes), which represents mock to external libraries; `AmazonEC2Mock` (Camel) and `MockGitHub` (Retrofit) are examples in this category. The third most frequent is web service (93 mock classes), that is, classes that are mock to web actions. Examples of web service include: `MockClientHttpResponse` (Spring Framework) and `MockHttpResource` (Elasticsearch). Other categories are Test support (11%, 67 mock classes), Native Java libraries (9%, 53 mock classes), and Network service (7%, 42 mock classes); notice that we do not find any mock class to the database category.

Table V
MOCK CLASS CATEGORIES.

Category	#	%
Domain object	211	35
External dependency	138	23
Web service	93	15
Test support	67	11
Native Java libraries	53	9
Network service	42	7
Database	0	0
Total	604	100

Previous studies report that domain objects, external dependencies, and web services are also among the most mocked categories when using mocking frameworks [2], [3] (*e.g.,* domain object is the most frequent in both our research and the mentioned studies). Another similarity with our results is regarding the least frequent categories: test support and native Java libraries are rarer in both analyses. Thus, our results complement the research literature by showing that, overall, developers tend to mock more and less frequently the same dependencies no matter they are mocking with the support of mocking frameworks or mock classes.

However, there are also some differences with respect to the results found in the mocking frameworks: we are not able to find mock classes in the database category, while in mocking frameworks this category is recurrent [2], [3]. We acknowledge that those differences regarding the database category may be because distinct projects on different domains are analyzed. Moreover, we are more strict in our definition of database category (*i.e.,* classes that are mock to SQL/NoSQL database library) to avoid subjectivity. Lastly, another difference is the network service category, which is present in our study and not in the ones about mocking frameworks [2], [3].

To complement this analysis, Table VI presents the top-3 most frequent terms on each category. As we can notice, categories as web service and network service are dominated by terms that closely relate to their purposes. For instance, in the case of web service, the term *http* refers to the web protocol, *request* is a part of how the http protocol makes communication (request/response), and *mvc* is related to the architectural pattern Model-View-Controller. Similarly, the network service category includes terms as *channel*, *transport*, and *connection*, which are typical in the network domain. The categories domain object, native Java libraries, external dependency present somehow more generic related terms. This may be related to the fact that these categories are broader. Domain object is specific to each project, and those may be as varied as there are classes within the project. The same happens to native Java libraries and external dependencies, which are likely to be project-specific.

Summary RQ1: Mock classes are often created to emulate domain objects, external dependencies, and web services. Those categories are essentially the same types created by developers when using mocking frameworks [2], [3].

Table VI
TOP-3 MOST FREQUENT TERMS PER MOCK CLASS CATEGORY.

Category	Frequent Terms
Domain object	Script, Filter, Factory
External dependency	Client, Amazon, Service
Web service	Http, Request, Mvc
Test support	Factory, Test, Mvc
Native Java libraries	Context, Config, Runnable
Network service	Channel, Transport, Connection

B. RQ2 (Design) How are mock classes designed?

We start by analyzing structural details of the mock classes, such as class extension and interface implementation. We then analyze the visibility of the mock classes and the number of methods in mock classes as compared to regular classes.

Class extension and interface implementation. Table VII details the number of mock classes that are derived from class extension and interface implementation. Our first observation is that class extension is more frequent than interface implementation: 54.9% (332 out of 604) of the mock classes extend other classes, whereas 46.7% (282 out of 604) implement interfaces; only 7.5% (45 out of 604) do not extend classes nor implement interfaces. Notice, however, that this rate may vary per system: in Lucene-Solr, for instance, 82.6% of the mock classes are about extensions, while only 21.2% are interface implementation. On the other hand, in Dubbo, interface implementation is more common (82.8%) than class extension (17.2%).

Visibility. Table VIII summarizes the visibility of the studied mock classes. We can observe that the majority of the mock classes (68.4%, 413 out of 604) are *public*, thus, they are visible to all classes and can be reused. The *protected* visibility is the least frequent, presented in only 7 classes (1.2%). Next, 13.1% (79 out of 604) of the mock classes have the *package* visibility, that is, they are visible only within their own packages. Finally, 17.4% (105) of the mock classes are *private*, therefore, they are only accessed in their own classes. Interestingly, although reuse can be considered an important advantage for creating mock classes [15], the most restrictive controlling accesses (*package* and *private*) correspond to about 30% of the mock classes.

Number of methods. Table IX presents some statistics about the number of methods in mock classes; for comparison, we also present the number of methods in randomly selected regular classes.[16] Although we find some variation among the investigated projects, overall, both mock and regular classes have 3 methods on the median. Figure 1 contrasts the distribution of the number of methods in both groups. By applying the Mann–Whitney test, we confirm both distributions of the number of methods is equivalent in mock and regular classes (the difference is not statistically significant, with *p-value*=0.11).

Summary RQ2: Mock classes often extend other classes or implement interfaces. Around 70% of the mock classes are public and can be reused, while 30% have restrictive visibility (*i.e.,* private or package). Overall, mock and regular classes have the same number of methods.

C. RQ3 (Usage) How are mock classes used by developers?

In this research question, we focus on the client-side of the mock classes. For this purpose, we analyze the Boa dataset [11] to assess mock classes. After filtering out possible false positives, we detect 6,444 mock classes that are used 147,433 times (see Section III-C). This suggests that the adoption of mock classes is a common practice.

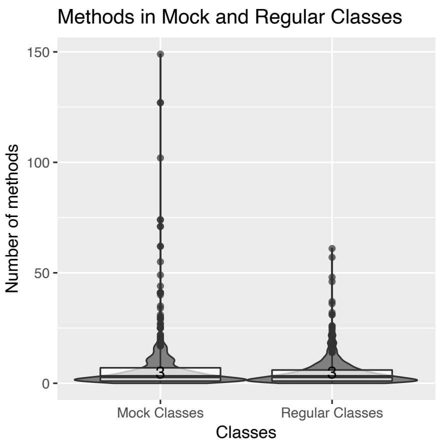

Figure 1. Distribution of the number of methods in mock and regular classes.

We now assess the 604 mock classes analyzed in this study to better understand how they are used in this large dataset by external clients. As presented in Table X, we find that 128 out of the 604 mock classes are used 52,079 times. Web service is the most consumed category, being used by 24,022 clients (46%). Next, the second and third most consumed categories are domain object and external dependency, adopted by 11,879 (23%) and 7,624 (15%) clients, respectively. The least consumed categories are native Java library, test support, and network service.

Figure 2 details the distribution of the usage per category. For example, on the median, the web service mock classes are used by 66 clients, while the domain object ones are used by 26 clients. The highest usage happens in native Java libraries (75 clients) and the lowest one happens in the category test support (10 clients). Interestingly, although only 18 mock classes to native Java libraries are used in this dataset, they are highly consumed.

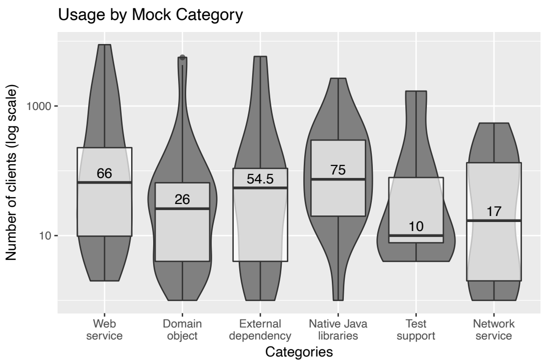

Figure 2. Usage of mock classes according to their categories.

[16]We randomly selected 604 classes from the target projects following the same distribution of the mock classes.

Table VII
CLASS EXTENSION AND INTERFACE IMPLEMENTATION IN THE MOCK CLASSES.

Project	**Mock Classes**	**Class Extension**	**%**	**Interface Implementation**	**%**	**None**	**%**
Elasticsearch	138	88	63.8	56	40.6	13	9.4
Spring Boot	33	8	25.0	18	54.5	8	24.2
Guava	12	4	33.3	8	66.7	0	0
OkHttp	1	0	0.0	1	100.0	0	0
Spring Fw.	127	50	39.4	74	58.3	14	11.0
Retrofit	4	1	25.0	1	25.0	2	50.0
Dubbo	58	10	17.2	48	82.8	1	1.7
Lucene-Solr	132	109	82.6	28	21.2	2	1.5
Apache Camel	99	62	62.6	48	48.5	5	5.1
Total	604	332	54.9	282	46.7	45	7.5

Table VIII
VISIBILITY IN THE MOCK CLASSES.

Project	**Mock Classes**	**Public**	**%**	**Protected**	**%**	**Package**	**%**	**Private**	**%**
Elasticsearch	138	84	60.1	5	3.7	17	12.3	32	23.2
Spring Boot	33	15	45.5	0	0	15	45.5	3	9.1
Guava	12	2	16.7	0	0	2	16.7	8	66.7
OkHttp	1	1	100.0	0	0	0	0	0	0
Spring Fw.	127	92	72.4	0	0	9	7.1	26	20.5
Retrofit	4	2	50.0	0	0	2	50.0	0	0
Dubbo	58	54	93.1	0	0	2	3.4	2	3.4
Lucene-Solr	132	77	58.3	2	1.5	24	18.2	29	21.9
Apache Camel	99	86	86.9	0	0	8	8.1	5	5.1
Total	604	413	68.4	7	1.2	79	13.1	105	17.4

Table IX
NUMBER OF METHODS IN THE MOCK AND REGULAR CLASSES.

Project	**Methods in Mock Classes**			**Methods in Regular Classes**		
	mean	**median**	**std. dev.**	**mean**	**median**	**std. dev.**
Elasticsearch	4.6	2	5.1	5.5	4	6.3
Spring Boot	2.8	2	2.6	3.2	2	3.9
Guava	8	5.5	7.4	2.8	2	2.4
Spring Fw.	12.4	6	20.0	5.1	2	8.3
Dubbo	6.1	2.5	8.1	7.4	4.5	9.5
Lucene-Solr	3.7	2	5.6	5.3	3	5.4
Apache Camel.	10.9	4	21.0	4.9	3	6.8
All	7.2	3	13.8	5.3	3	6.9

Table XI presents the top-10 most consumed mock classes. Classes `MockHttpServletRequest` and `MockHttpServletResponse` are the most used ones; they are both provided by Spring to facilitate web testing. The third one is `MockEndpoint`, which is provided by Apache Camel for testing routes and mediation rules using mocks. Half of the top-10 most consumed classes are about web services. Together, the top-10 mock classes are used by 39,693 out of 52,079 clients (76%), while the top-50 are responsible for almost all client usage (97%).

Summary RQ3: Mock classes are largely consumed by client projects to support testing; web services are the most emulated dependencies. The usage is very concentrated on a few classes: 10 classes are used by 76% of the clients, while 50 classes are consumed by 97%.

Table X
MOST FREQUENT MOCK CATEGORIES IN THE BOA DATASET.

Category	#Classes	#Clients	%	
Web service	35	24,022	46	▮
Domain object	39	11,879	23	▮
External dependency	14	7,624	15	▮
Native Java libraries	18	5,339	10	▮
Test support	7	1,899	4	▮
Network service	15	1,316	2	▮
Total	128	52,079	100	■

Table XI
MOST FREQUENT MOCK CLASSES IN THE BOA DATASET.

Mock Class	Mock Category	#
MockHttpServletRequest	Web service	8,846
MockHttpServletResponse	Web service	6,104
MockEndpoint	External dependency	5,821
MockAnalyzer	Domain object	5,628
MockTokenizer	Domain object	4,288
MockServletContext	Native Java libraries	2,675
MockMvc	Web service	1,822
MockContext	Test support	1,699
MockResponse	Web service	1,575
MockWebServer	Web service	1,235
Top-10	-	39,693
Top-50	-	50,709

V. DISCUSSION AND IMPLICATIONS

A. Assessing mock classes

Mock objects are often used to support software testing [1], [4]–[7], however, there is a surprising lack of research studies on that topic. Although mock classes are typically provided by large and popular projects (as the ones assessed in this research, *e.g.,* Elasticsearch, Spring Framework, and Lucene-Solr), actual mock studies are limited to the context of mocking frameworks [2], [3], [8], [10]. *Therefore, we contribute to the software testing literature with a novel study about mock classes and their usage in order to complement existing research in the context of mocking frameworks.*

B. Novel empirical data on mock classes

Our study reveals new data about mock classes. We find that they are concentrated on the categories domain object, external dependency, and web service (which is similar to the mocks created by frameworks). Moreover, we also present several structural information about the mock classes. For example, they mostly rely on inheritance and interface implementation, that is, mock classes are rarely standalone. They are typically public and can be reused but private mock classes are not rare. Also, mock classes have the same number of methods of regular classes, therefore, their overall maintenance effort is equivalent. *Thus, we reveal novel quantitative and qualitative empirical data about the creation of mock classes, which can guide practitioners in charge of maintaining them. We show that mock classes are over-concentrated on certain tasks, are often part of a hierarchy, are mostly public, and are not different from regular classes regarding number of methods.*

C. Reuse and lack of visibility of the mock classes

One of the benefits of creating mock classes by hand is their reuse power [15]. For example, a single mock class provided by system X can be used to support the creation of test cases in X itself and in the clients of X. Indeed, as we presented in RQ3, mock classes may have thousands of clients. However, we find that about 30% of the analyzed mock classes have *package* or *private* visibility. That is, their reuse is very limited: they are visible only within their own packages or classes. Thus, the lack of reusability on almost one-third of the mock classes is a surprising result. As those private mock classes are not intended to be reused, they are straightforward candidates to be mocked with frameworks; in this case, the maintenance effort would be smaller as less mock classes would be available. *Thus, we shed light on the over creation of private mock classes, which may be harmful to the overall project maintainability. This can drive future research agenda on techniques to detect superfluous mock classes that can be created with mocking frameworks.*

D. Widespread usage of the mock classes

We also find that mock classes are employed to a much larger extent. When analyzing the Boa dataset [11], we detected that mock classes are consumed by thousands of open-source software projects. That is, the usage of mock classes is not restricted to the target projects of this study, but it seems to be widespread. *Thus, practitioners who maintain mock classes (as the ones assessed in this research) should be aware of the importance of these classes to their ecosystems. During maintenance, mock classes should be changed with care because a large number of client projects can be impacted.*

E. Reasons for using mock classes and mocking frameworks

Overall, the frequency of mock categories detected in this study is similar to those found by earlier studies on mocking frameworks [2], [3]. For example, domain objects, web services, and external dependencies are common in both studies, while test support and native Java libraries are rarer. This suggests that independently the way developers are mocking dependencies (*i.e.,* either via mock classes or mocking frameworks), the overall goal is the same. Indeed, as presented in Table I, most of the analyzed projects in this study use both solutions to mock dependencies. For example, Elasticsearch has 524 types mocked with the mocking framework Mockito and 138 mock classes. Thus, a question arises: when do developers rely on mock classes instead of mocking frameworks, and vice-versa? A natural explanation is that the mock classes could be intended to be consumed by external clients to facilitate their tests. In fact, other reasons may arise in favor of one solution over the other. *In this way, we shed light to further investigation on the specific reasons practitioners use to create mock classes by hand and with the support of mocking frameworks.*

VI. Threats to Validity

In this section, we detail the threats that may affect the validity of the study and how they are handled.

A. Focus on libraries instead of end-user products

The projects we selected in this study are libraries or frameworks that are typically used as dependencies by end-user software. For example, Spring Boot is a framework used to build web applications, and not a final application itself. Indeed, libraries and frameworks are really important to support software development nowadays, providing feature reuse, improving productivity, and decreasing costs [18]–[22]. However, we recognize they do not comprehend the reality of those end-user products when using mock classes, so this needs to be taken into account when interpreting our categories of mock classes. Notice that end-user software products are better represented in the results presented in RQ3, in which we assessed millions of Java projects with the support of the Boa platform [11]. In this case, web services were the most common mocked dependencies.

B. Manual classification

Mock classes in our study were manually classified by the first author of the paper, who is a software engineer with 8 years of experience in embedded, desktop, and web development. Many of them were classified based on strong terms present in their names (*e.g.,* Mvc, Http, Rest, TCP, GitHub, etc). In cases the names were not clear (or in cases of doubts), the author relied on additional artifacts, and carefully analyzed documentation or inspected the mock source code to infer the category (see Section III for more details). Thus, like any other manual classification, it is subjected to error and bias. However, an evidence that may minimize this threat is that the frequency of mock categories detected in this study is similar to those found by earlier studies on mocking frameworks [2], [3], that is, domain objects, web services, and external dependencies are frequent on both research studies, while test support and native Java libraries are less common.

C. Lack of mock classes categorized as Database

In the previous study about mock objects and mocking frameworks in Java [2], [3], a database mock is defined as one "*that interact with an external database. These classes can be either an external library [...] or a class that depends on such external libraries [...]*". In our study, we were more strict in the definition of database: we only stated a mock class to be in the database category when it was directly linked to a SQL/NoSQL database library. We were more strict due to two reasons. First, we found the original definition a bit subjective and flexible. Second, one of the projects in our study, Elasticsearch, is itself a NoSQL database, thus, to some extent, all mock classes in this system could be classified in the database category, and, of course, this would not be desirable. Therefore, according to our criteria, we found no mock classes for databases.

D. Identifying mock classes

Our selection criterion for identifying mock classes is that they should contain the string "mock" in the class name. While our findings show that there is plenty of mock classes that follow this guideline, there is the possibility we lose track of classes that are used as mocks but do not follow it.

E. Generalization

We analyzed 604 mock classes provided by several popular and real-world Java software systems. For instance, the projects Elasticsearch, Spring Boot, and RxJava have over 40k stars, thus, they are among the most popular in the Java ecosystem, as measured by Github. Moreover, in our third research question, we searched for the usage of mocks in millions of projects with the Boa platform [11]. Despite these observations, our findings — as usual in empirical software engineering — may not be directly generalized to other systems, as commercial ones with closed source and implemented in other programming languages.

VII. Related Work

Spadini *et al.* state that mock objects are common when testing software dependencies and their use is supported by a variety of frameworks in several programming languages [2]. The authors present that despite this practice being common there is a lack of empirical knowledge on how and why practitioners use mocks. To this aim, the study detects classes that are mocked with the Mockito framework in Java projects and categorize them. Moreover, the authors interview developers to understand the reasons for mocking dependencies and the main challenges they face. The study finds that the dependencies that are most mocked are those that make testing difficult and that classes that are in complete control of the developers are least mocked; the challenges are related to technical issues such as coupling between the mock and the production code. The authors then extended their study with investigations about mocks introduction and evolution and expanded code quality metrics evaluation [3]. The new research confirms the findings of their previous study [2] in the sense developers tend to mock dependencies that make tests more difficult, and show that mocks usually evolve together with the test classes, being added at the beginning of their history and changing accordingly. In our study, we focus on *mock classes* instead of mock objects that are mocked via mocking frameworks like Mockito. Therefore, the results presented in this paper complement previous ones with respect to mocking frameworks.

Mostafa and Wang study mocking frameworks in software testing, showing their wide usage in this discipline and calling for more studies on the topic [8]. Arcuri *et al.* assess the role of mock objects in automatic unit test generation, stating that its use in their EvoSuite[17] tool helps increasing branch coverage and fault detection [10].

There is a vast technical literature about mock objects [1], [4]–[7]. Meszaros provides an introduction and discussion about mock objects, presenting a taxonomy based on their purpose and usage [1]. This taxonomy is further explored and discussed by Martin Fowler [14], [23]. The creation of manual mock classes versus the automatic creation via frameworks is also discussed by Martin Fowler, stating that developers write their own mocks to improve reuse, project design, and performance due to not using reflection [24]. In contrast, some authors observe a disadvantage in using mocks. For instance, Elliot discusses how mocks can be code smells by relating them to tight coupling, a symptom of code smells itself, particularly in the context of functional programming [25]. The author states that the drive for mocking in unit tests is to achieve complete test coverage but there is a flawed decomposition strategy, which causes the need for mocks.

The discussion about the relevance of software testing is not new. It is commonly stated the importance of testing across unit, integration, and system stages [26]. In this context, the literature emphasizes that testing code (as any other code) should be maintained as software evolve [12]. This way, issues related to code maintenance as code smells [13] are valid in tests as well, as test smells [27]–[30]. Therefore, if one considers that mock classes are part of the tests themselves, maintaining tests implies maintaining the mock classes as well.

Although few research studies assess mock objects, they focus on the perspective of mocking frameworks. Our study assesses the mock classes created by hand to support testing, thus we complement the existing literature on mock objects.

VIII. Conclusion

In this paper, we presented an empirical study to assess mock classes. We analyzed 12 popular Java projects and detected 604 mock classes. We proposed research questions to assess the content, design, and usage of mock classes. By applying quantitative and qualitative analysis, we found:

- Mock classes are often created to emulate domain objects, external dependencies, and web services.
- Mock classes are often part of a hierarchy, are mostly public, and are not different from regular classes regarding number of methods.
- Mock classes are largely consumed by client projects to support testing, particularly to emulate web services.

Based on our results, we provided insights and practical implications for researchers and practitioners. We reveled novel empirical data about mock classes, which can guide practitioners dealing with mock classes; we brought to light the over creation of private mock classes, which can be harmful to reuse; and we presented that the consumption of mock classes is widespread, thus, they should be maintained with care because client projects can be largely impacted.

As future work, we plan to better investigate the reasons for developers to rely on mock classes instead of mocking frameworks. We also plan to take a deeper look at the semantics of the mock classes to refine their classification and provide insights on how they mock dependencies. Finally, another goal is to look for mock classes in other programming languages than Java (*e.g.,* Python and JavaScript) to assess the concerns of other software ecosystems.

Acknowledgment

This research is supported by CAPES and CNPq.

References

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

[2] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices," in *International Conference on Mining Software Repositories*, 2017, pp. 402–412.

[3] ——, "Mock objects for testing java systems," *Empirical Software Engineering*, vol. 24, pp. 1461–1498, 2019.

[4] M. Feathers, *Working Effectively with Legacy Code*. Prentice Hall Professional, 2004.

[5] S. Freeman and N. Pryce, *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[6] R. Osherove, *The Art of Unit Testing: With Examples in. Net*. Manning Publications Co., 2009.

[7] H. Percival, *Test-driven development with Python: obey the testing goat: using Django, Selenium, and JavaScript*. O'Reilly Media, Inc., 2014.

[8] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *International Conference on Quality Software*, 2014, pp. 127–132.

[9] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *Workshop on Automation of Software Test*. IEEE, 2009, pp. 149–153.

[17] www.evosuite.org

[10] A. Arcuri, G. Fraser, and R. Just, "Private API access and functional mocking in automated unit test generation," in *International Conference on Software Testing, Verification and Validation*, 2017, pp. 126–137.

[11] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *International Conference on Software Engineering*, 2013, pp. 422–431.

[12] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[13] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[14] ——. (2007) Mocks aren't stubs. [Online]. Available: https://martinfowler.com/articles/mocksArentStubs.html

[15] R. Martin. (2014) When to mock - the clean code blog. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2014/05/10/WhenToMock.html

[16] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *International Conference on Software Maintenance and Evolution*, 2016, pp. 334–344.

[17] H. Silva and M. T. Valente, "What's in a GitHub star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[18] S. Moser and O. Nierstrasz, "The effect of object-oriented frameworks on developer productivity," *Computer*, vol. 29, no. 9, 1996.

[19] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *International Computer Software and Applications Conference*, 2009, pp. 287–292.

[20] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *International Conference on Software Maintenance*, 2012, pp. 378–387.

[21] G. Menezes, B. Cafeo, and A. Hora, "Framework code samples: How are they maintained and used by developers?" in *13th International Symposium on Empirical Software Engineering and Measurement*, 2019, pp. 1–11.

[22] C. Lima and A. Hora, "What are the characteristics of popular apis? a large scale study on java, android, and 165 libraries," in *Software Quality Journal*, vol. 28, 2020, p. 425–458.

[23] M. Fowler. (2006) bliki: TestDouble. Library Catalog: martinfowler.com. [Online]. Available: https://martinfowler.com/bliki/TestDouble.html

[24] R. Martin. (2014) When to mock. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2014/05/10/WhenToMock.html

[25] E. Elliott. (2017) Mocking is a code smell. Library Catalog: medium.com. [Online]. Available: https://medium.com/javascript-scene/mocking-is-a-code-smell-944a70c90a6a

[26] E. Weyuker, "Testing component-based software: a cautionary tale," *IEEE Software*, vol. 15, no. 5, pp. 54–59, 1998.

[27] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *International Conference on Automated Software Engineering*, 2016, pp. 4–15.

[28] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *International Conference on Software Maintenance and Evolution*, 2017, pp. 1–12.

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

[30] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.