An Empirical Study on the Usage of Mocking Frameworks in Apache Software Foundation

Lu Xiao[a], Keye Li[a], Erick Lim[a], Xiao Wang[a], Chenhao Wei[a], Tingting Yu[b] and Xiaoyin Wang[c]

[a]Stevens Institute of Technology, Castle Point Terrace, Hoboken, NJ 07030, United States
[b]University of Cincinnati, 2600 Clifton Ave, Cincinnati, OH 45221 United States
[c]University of Texas at San Antonio, 1 UTSA Circle, San Antonio, TX 78249 United States

ARTICLE INFO

Keywords:
software testing, mocking frameworks, Apache open source projects

ABSTRACT

Mocking frameworks provide convenience APIs, which create mock objects, manipulate their behavior, and verify their execution, for the purpose of isolating test dependencies in unit testing. Mustafa and Wang studied the usage of mocking frameworks based on 5000 GitHub projects. Our study advances this understanding based on 193 Apache projects, which have different "demographic" features from the 5000 GitHub projects—the average LOC of projects is 25K for GitHub and 165K for Apache. Apache projects, such as Hadoop and Cassandra, are widely used in different domains and systems. We found that 67% of the Apache projects use at least one mocking framework, in comparison to the 23% on GitHub. Mockito and EasyMock are the most popular mocking frameworks—used in more than 90% Apache projects—this is consistent for the GitHub projects. It is also worth noting that 12% Apache projects use Spring Framework for mocking web services, which is not found in GitHub projects. Apache developers leverage the mocking APIs mostly (61.1%) for replacing external library classes; in comparison, a smaller portion (39.4%) of mock objects in GitHub projects are for library classes. Our study provides valuable empirical experience for practitioners regarding when and how mocking frameworks are used in practice.

1. Introduction

Software testing is the process of verifying and validating the functional and non-functional attributes of a software system. Unit testing, the most fundamental phase of software testing, targets at a software system as units, typically as methods [1, 2, 3, 4, 5]. A unique challenge in unit testing results from the inter-dependencies among the units [1, 6]. That is, one unit usually has dependencies to other units in the system, as well as to external systems or third-party libraries. Therefore, it is inappropriate, and often impractical, to test a system as completely separate units without considering their dependencies. For example, a unit of function under test (FUT) may depend on an external database for data storage. This dependency hinders the testing of the FUT. For instance the database may not be deployed and ready for use; connecting to the database may not be affordable in continuous testing; and bugs in the database may cause interference to the testing and debugging of the FUT.

In order to overcome these challenges, practitioners devised a mechanism called mocking, which replaces test dependencies of the FUT by creating mock objects [7, 8]. That is, developers create a faked object and control its behavior to mimic the behavior of a dependency for the testing purpose. For example, developers may leverage the file system with hard-coded data items to replace the real database [9]. Mocking helps to isolate dependencies and enforce true "unit" testing. Developers can test the system as separate units in parallel, without having to wait for each other. If the dependency is to an external system, such as database or an http server, mocking helps to avoid the long waiting time to access external resources, which could be exorbitant in unit testing using the continuous testing and integration flow. Furthermore, isolating test dependencies through mocking can avoid bug interference in debugging the FUT. Without mocking the dependencies, bugs outside of the FUT can also cause test failures, making debugging more confusing and less efficient.

There exist dedicated mocking frameworks, such as Mockito, EasyMock, and PowerMock, for facilitating mocking in Java projects. They provide convenient APIs for creating mocking objects, manipulating the behaviors of the mocking objects through method-stubbing, and verifying the execution status and interactions of the mock objects. Despite the various benefits of using these frameworks for mocking, there are also debates regarding their usage, focusing on when and how mocking frameworks should be used. For example, one of the main concerns is the raised bar for developers to contribute in open source projects, and the lack of sufficient coverage in the current curriculum of software testing. We found in our previous study [10] that developers sometimes turn to inheritance as a way for mocking since it is more intuitive for developers who are not familiar with mocking frameworks.

There currently is limited knowledge regarding whether and how mocking frameworks are used in practice. Related empirical experience can benefit practitioners in learning and adopting mocking frameworks in their projects. Mostafa and Wang [11] conducted an empirical study on how mocking frameworks are used in the github community. Their

lxiao6@stevens.edu (L. Xiao); kli32@stevens.edu (K. Li); elim1@stevens.edu (E. Lim); xwang97@stevens.edu (X. Wang); cwei7@stevens.edu (C. Wei); tingting.yu@uc.edu (T. Yu); xiaoyin.wang@utsa.edu (X. Wang)
ORCID(s):

Electronic copy available at: https://ssrn.com/abstract=4100265

study focused on 5000 software projects from github. They focused on three research questions:

- *RQ1: How popular are mocking frameworks? Are developers trying to mock most or all of dependencies?* They found that among the 5000 projects, 2,046 has at least one test class. And 459 (23%) uses at least one mocking framework. Projects using mocking frameworks are larger in size, with 16KLOC as the median size, than those which do not use mocking—with 8.4KLOC median size. frameworks. Overall, 17% of dependencies are mocked by the software testers—indicating only a small portion of all dependency classes are mocked.
- *RQ2: What features of mocking frameworks are most frequently used in the testing of open source software projects?* Mockito and EasyMock are the most widely used. The top four most popular frameworks used by projects on github include, Mockito, EasyMock, JMock, and JMockit. Mostafa and Wang's study [11] investigated the most popular APIs from Mockito and EasyMock. They found that software testers use advanced APIs, such as *verify* and *spy*, for specifying and verifying the interactions between the FUT and the mock objects, instead of creating simple test stubs or fake objects.
- *RQ3: What types of dependencies developers tend to mock?* They found that software testers tend to use 60% of the mock objects for replacing source code classes and the remaining 40% for library classes. The most frequently mocked library classes are for handling HTTP requests /responses, and content repositories.

This study answers the above questions focusing on all Java projects hosted on the Apache Software Foundation [12]. Apache is one of the largest open source communities in the world, hosting more than 350 projects implemented in different languages, including Java, Python, C++ etc.. A total of 243 projects are implemented in Java. The projects hosted on GitHub and Apache have very different "demographics". Apache projects tend to be larger-scale and more complicated. The average LOC of projects on GitHub is 25K, while the average LOC on Apache is 165K. Many Apache projects, such as Hadoop and Cassandra, are widely used in different domains and systems, having profound impacts to the society. While, many of the GitHub projects are individually owned. We are motivated to reveal whether and how mocking frameworks are differently in Apache projects. This can help practitioners to achieve more comprehensive understanding of how mocking frameworks are used in practice, based on the different nature of projects.

Our work also extends Mostafa and Wang's study [11] by adding a fourth research question. Namely,

- *RQ4: Do developers always use a mocking framework when mocking is needed?* If not, this may point to opportunities where a mocking framework should be properly used or potential limitations of existing mocking frameworks. Such understanding informs practitioners when a mocking framework may not be sufficient.

The rest of this paper is organized as follows. Section 2 discusses background information. Section 4 introduces our study process. Section 5 elaborates the research questions and our findings. Section 5.5 compares our findings with that of Mostafa and Wang's study. Section 6 discusses limitations and threat to validity. Section 7 discusses related work. Section 8 concludes this study.

2. Background

2.1. Motivating Example for Mocking

Figure 1 illustrates the concept of test dependencies in a real-life scenario. In an E-commerce system, the *Customer Service* module is responsible of providing various services for customers, such as subscribing a new customer to the system and sending an email confirmation. In fulfilling its functions, it send requests to and receive responses from a web server, which communicates with a SQL Database.

When testing the function of *Customer Service* module as a unit, the tester must consider its dependencies to the *Web Server* as well as the *SQL Database*. If these two dependencies are not available, , e.g. the server and the database are not deployed, the testers cannot test the functions of *Customer Service* easily. If the tested function involved a large amount of network data transmission, running the test cases for *Customer Service* could lead to long waiting time in the Continuous Integration Cycle. Finally, if the *Web Server* or the *SQL Database* contains bugs, these bugs could interfere the test cases of *Customer Service*—requiring extra effort in the debugging of the *Customer Service* module.

Mock objects are designed to address the above challenges by isolating the function under test from its dependencies. For example, as illustrated in Figure 2, in the unit testing of the *Customer Service* module, the tester can create a fake server to replace the dependency on the server. More specifically, instead of accessing a real web server, the *Customer Service* talks to a fake server, which mocks the behavior of the real server in a controlled way, just for the purpose of testing. For example, the fake server may always return *true*, indicating the request was successfully received and processed. As such, the tester can focus on the function under test with the help of the mock server.

2.2. Mocking Frameworks

There is a number of mocking frameworks which are dedicated for creating, manipulating, and verifying mock objects in unit testing. These frameworks provide a variety of convenient APIs for three different aspects in mocking: 1) creation of mock objects; 2) manipulation of the behavior of mock objects; 3) verification of the interactions with and status of mock objects. There are different mocking frameworks for different programming languages,

Electronic copy available at: https://ssrn.com/abstract=4100265

Figure 1: Test Dependencies on Web Server and SQL Database

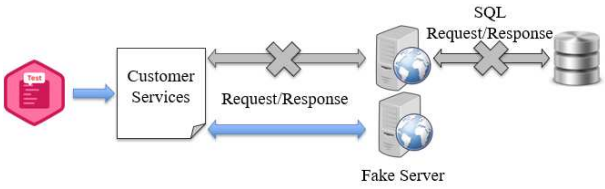


Figure 2: Dependency Isolation by an Mock Object

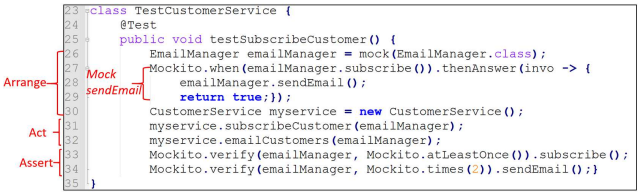


Figure 3: Mocking by Mockito

such as Mockito [13], EasyMock [14], PowerMock [15] and SpringframeworkMock [16] for Java; Mock [17] for Python; NMock [18], Moq [19] for C#. These frameworks are widely used in software projects to ease the process of unit testing [20, 21, 22].

Figure 3 is the implementation of the motivating example in the previous subsection, implemented following the syntax of Mockito. The function under test (FUT) is the *CustomerService*. The FUT depends on the *EmailManager* deployed on an external web server for sending emails to customers. In this example, we create a mock object of *EmailManager* (line 26). And, we control the behavior of *EmailManager* by stubbing its method *subscribe* and *sendEmail* (line 27 to 29). When acting the FUT, we pass the mock object to isolate test dependency (line 31 and 32). Finally, we use the verify function to check the execution of the two stubbed methods (line 33 and 34).

3. Research Questions

We ask the following research questions to understand the usage of mocking frameworks in Apache projects.

- *RQ1: How popular are mocking frameworks in Apache projects?* We analyzed the usage of the top four most popular mocking frameworks in Apache Java projects. We investigate the portion of test files that using mocking frameworks and the number of mock objects in each test files to understand if mocking frameworks are widely used in test development of Apache projects.
- *RQ2: What are the most frequently used mocking APIs in the testing of Apache projects?* We investigate the top 10 most frequently used mocking APIs in the two most popular mocking frameworks to understand how developers use mocking frameworks in Apache projects.
- *RQ3: What types of dependencies developers tend to mock in Apache projects?* We analyzed the most frequently mocked classes to understand whether developers tend to mock external library classes or the classes in their own projects.
- *RQ4: Do developers always use a mocking framework when mocking is needed?* We analyzed the number of files that contain keyword *Mock* in their file name or path but do not have dependencies to any mocking frameworks. The goal is to understand are there any files been used as mock object without using any mocking frameworks. These files may indicate a sub-optimal mock implementation and can be refactored by using mocking frameworks.

4. Study Process

This study focuses on the open source projects hosted on Apache Software Foundation. There are a total of 246 java projects. In order to answer the above research questions, our study process contains five main conceptual steps:

1. Step 1: Basic Data Collection: We collect basic project information of the study subjects.
2. Step 2: Mock Frameworks Identification: We extract imported libraries in each project, and manually verify whether a mocking framework is used.
3. Step 3: Mock API Analysis. We extract and analyze the frequently used mocking APIs from different mocking frameworks.
4. Step 4: Mock Dependency Analysis. We investigate whether the mock objects are internal to a project, or external libraries or resources.
5. Step 5: Sub-optimal Mock Identification. We identify cases where developers leverage the concept of “mocking" without replying on a mocking framework.

We have publicized our scripts and intermediate data of each step here `https://github.com/RedRoach51/MockResearch`.

4.1. Step 1: Basic Data Collection

Firstly, we collect the source code of each project by cloning their git repository. We find that 33 projects do not have a linked Git repository. In addition, in 4 projects, we do not identify any test cases. Next, we import each project to *eclipse JDT* [23], which resolves the bindings among the software entities to prepare for the following-up analysis. We

Electronic copy available at: https://ssrn.com/abstract=4100265

Table 1
Basic Information of Mock Usage Empirical Study Subjects

Basic Information	Avg.	Med.	Max.	Min.
#Java Files	1537	738	20760	17
#Test Files	524	189	11064	1
LOC	165443	76517	1574022	993
#Developers	110	44	2488	2
#Versions	56	36	584	0

cannot properly configure and compile 16 projects. Therefore, our dataset actually contains a total of 193 projects. We collect some basic measurements of these 193 projects, including the number of Java source files, the number of test files, the total LOC, the number of developers, and the number of available versions. Note that the number of test files is counted based on the import of *Junit* library [24]. If a *.java* file *imports* the *Junit* APIs, we consider it as a test file. Table 1 summarizes the average, median, maximal, and minimal of these measurements of the 193 projects.

4.2. Step 2: Mock Framework Identification

If a software project uses a mocking framework, it has to import the related APIs. Thus, in order to investigate whether the 193 projects use exiting mocking frameworks, and what are these frameworks, we extract all the API calls from each test file in a project. We leverage the eclipse JDT to resolve the bindings among software entities to retrieve the full name space of each API. Next, we use a simple key word, "mock", to search for all potentially related framework names. To the best of our knowledge, the name space of well known frameworks all contains this keyword, such as mockito, easyMock, powerMock, etc.. Based on our observation, even if it is not in the name of the framework, it appears as part of the import name space. We use this method to search for potential mocking frameworks to avoid missing ones that we are familiar with. Finally, we manually review the retrieved name spaces that match the search keyword. For each identified item, we manually verify whether it is a mocking framework by searching its information online. For the confirmed mocking frameworks, we count and rank their popularity in the 193 projects. This helps us to reveal which mocking frameworks are most popular.

4.3. Step 3: Mock API Extraction and Analysis

In step 2, we have identified whether a project uses a mocking framework, and which framework(s) are used in each project. In step 3, we focus the most popular mocking frameworks. It is of low value to investigate the APIs of a uncommonly used mocking framework. Based on the confirmed mocking framework name spaces, we search for APIs calls that match the framework namespace. For example, Mockito's APIs all start with *org.mockito.* We rank all the APIs of each mocking framework based on their frequencies being used across all the projects. In RQ2, we focus on the top ranked APIs in the most popular mocking frameworks in Apache software foundation.

4.4. Step 4: Mock Dependency Analysis

In this step, we focus on the object being mocked—whether the mocked object is an external library or an internal function, In order to achieve this goal, we first need to understand what is the syntax for creating a mock object using different frameworks. For this, we carefully review the official documentation of each mocking framework, and curate the APIs that can be used for mock object creations. For instance, both Mockito and PowerMock uses mock(). By matching the mock object creation APIs, we identify all the dependencies being mocked in a project. Next, we retrieved the full name spaces of the mocked dependencies. If a name space is consistent with that of the project, it implies that the object is internal to the project. For example, in PDFBox [25], all the internal objects have this name space, "org.apache.PDFBox". In comparison, an external dependency has a different name space from the project. We assume that there may exist common external dependencies that developers need to mock, such as a database or a http server. Therefore, we further count the most frequently mocked external dependencies.

4.5. Step 5: Sub-optimal Mock Analysis

In step 5, we aim to reveal whether developers leverage the concept of mock without using any existing mocking frameworks. It is possible that developers create mock objects using an informal approach, such as based on inheritance or by creating the mock objects manually [10, 26]. We believe that these cases may point to sub-optimal implementation of mock, due to various factors, such as lack of related experience in using a mocking framework or even limitation with an existing framework.

The heuristic we leverage in this step is that, we identify test files with keyword "mock" in their names, but does not import any APIs from existing mocking frameworks. Once we identify such cases, we randomly select sample cases and review how developers use the concept of "mock" without a mocking framework. This helps us to understand gaps in open source developers' expertise and even gaps in existing mocking frameworks. We acknowledge that it is possible that developers may not always include the keyword "mock" in such cases. Thus, we may not be able to retrieve all related cases using the searching heuristic.

5. Study Results

5.1. RQ1: Popularity of Mocking Frameworks

We find that, in the 193 projects, 129 (67%) projects use at least one mocking framework. This indicates that mocking frameworks are commonly used in Apache projects.

In addition, Figure 4 shows the comparison of the LOC for projects that use a mocking framework vs. projects that do not use a mocking framework. The median size of the projects with mocking frameworks is 122K LOC, comparing to the median 77K LOC of all projects. This indicates that larger projects are more likely to use a mocking framework compared to smaller projects.

Electronic copy available at: https://ssrn.com/abstract=4100265

Figure 4: Size Comparison between Projects Using Mocking Frameworks with Projects Not Using Mocking Frameworks (Use All Metrics Data)

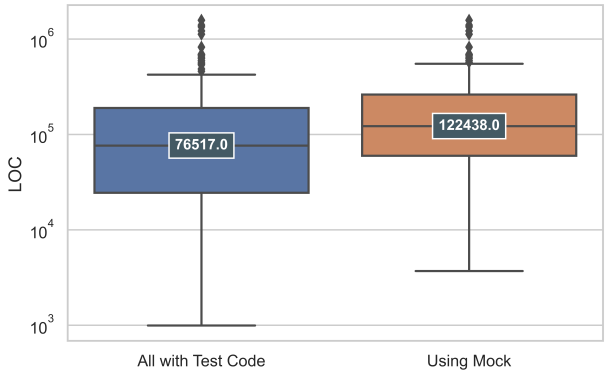


Table 2
Popularity of Mocking Frameworks

Mocking Framework	#Projects	% Projects	Acc. % Projects
mockito	99	77%	77%
easymock	36	28%	91%
powermock	16	12%	92%
springframework	16	12%	94%
Others	8	6%	100%
Total	129		
Multiple Mocking Frameworks	56 (40%)		

Table 3
Usage of Mocking Frameworks in Test Files

	Avg.	Med.	Max.	Min.
#Test Files	50.0	13.5	492	1
Proportion of Test Files	10.2%	6.05%	62.17%	0.03%

Table 2 shows the popularity of different mocking frameworks. As we can see, among the 129 projects using mocking frameworks, Mockito is the most popular—used in 95 (71%) projects. EasyMock and PowerMock, ranking in the second and the third place, are used in 27% and 12% projects, respectively. As shown in the last column, the top 4 most frequently used mocking frameworks together are used in 94% projects. It is worth noting that 54 (40%) projects use more than one mocking frameworks.

Furthermore, we investigate how intensively are the mocking frameworks used in the test files. The results are shown in Table 3. On average, 50.0 (10.2%) of test files in a project use mocking framework APIs. In other words, about one in ten test files in the Apache projects uses a mocking framework API. In most intense case, 62% of test files in a project uses mocking framework APIs.

Table 4
Proportion of Mocked Classes

	Avg.	Med.	Max.	Min.
#Mocked Classes	3.2	2	67	1
#Dependency Classes	23.1	19	277	1
Proportion of Mocked Classes	16.0%	12.1%	100%	0.7%

Table 5
Number of Mock Objects in Test Files

#Mock Objects	#Test Files (%)
1	1063 (27.7%)
2	721 (18.8%)
3	460 (12.0%)
4	373 (9.7%)
5	222 (5.8%)
6	174 (4.5%)
7	117 (3.1%)
8	106 (2.8%)
9	87 (2.3%)
10+	512 (13.4%)

Table 6
Top Ten Most Popular APIs In Mockito

API	Frequency	Description
when	13332	Specify a method and enable stubbing.
mock	10888	Create an empty mock object.
any	4485	Argument matcher that match any arguments.
verify	3609	Verify a specific behavior of the mock object.
thenReturn	3287	Stub the return value for a non-void method.
times	1536	Verify exact number of invocations for the methods of a mock object.
eq	1410	Check if argument value equals to a given value.
doReturn	1252	Stub the return value for any method.
spy	734	Create a spy object.
anyLong	596	Argument matcher that match any Long type arguments.

Summary: Mocking frameworks are widely used in Apache Java projects—67% projects use a mocking framework. On average, 10.2% test files in a project use mocking framework APIs. It is worth to note that larger projects are more likely to use mocking frameworks. Mockito is the most popular mocking framework—as 77% projects are using it.

5.2. RQ2: Usage of Mocking Frameworks

In RQ2, we analyze the number of mock objects created in a test file. Table 5 shows the percentage of test files that create from one to more than ten mock objects. We find that the majority (58.5%) of test files create three or less mocking objects. However, in 13.4% test files, developers may create more than ten mock objects. The implication is that there are often multiple dependencies that developers need to isolate for unit testing, and thus multiple mock objects are created.

We further analyze the most frequently used APIs. We focus on Mockito and EasyMock, since they are used in 91% of the 129 projects that use a mocking framework. Table 6 and Table 7 shows the top 10 most frequently used APIs in Mockito and EasyMock, respectively.

For Mockito, the most frequently used API is *when*; and for EasyMock, the second most frequently used API is *expect*. These two APIs are for manipulating the behavior of mock objects through method stubbing. This indicates that mocking frameworks are not just used to create dummy objects in a superficial way. Instead, developer control the

Electronic copy available at: https://ssrn.com/abstract=4100265

Table 7
Top Ten Most Popular APIs In EasyMock

API	Frequency	Description
createMock	2668	Create an empty mock object without default returns.
expect	2638	Specify a method and enable stubbing.
andReturn	1067	Stub the return value for a method returns anything but void.
replay	891	Switch mock object to replay mode.
verify	425	Verify a specific behavior of the mock object.
anyTimes	276	Expect a method of a mock object to be executed any times.
expectLastCall	264	Stub the behavior of void methods.
eq	238	Expects an object equals to the given value.
andStubReturn	216	Specific the default return for a method.
createNiceMock	5555	Create an empty mock object with default returns.
isA	138	Expects an object implementing the given class.

Table 8
Library Class Mocks

	Avg.	Med.	Max.	Min.
#Mocked Library Classes	25.4	9	622	0
Proportion of Library Classes	61.1%	65.2%	100%	0%

behavior of the mock objects through related APIs for serving the testing purposes.

APIs for creating mock objects, including *mock* in Mockito and *createNiceMock* and *createMock* in EasyMock, are also among the top 10 most frequently used APIs. The implication is that practitioners should start with these APIs with creating mock objects in the learning process, since they are most frequently used.

Finally, APIs for checking the execution status and interactions with mock objects are also in the top 10 most popular list. For example, *verify* and *times* are two advanced APIs in Mockito for verifying the certain behavior and invocation number associated with the mock objects. Similarly, the counter part APIs in EasyMock, namely, *anyTimes* and *verify*, are also frequently used.

Summary: The majority (58.5%) of test files create 3 or less mocking objects. Developers are not simply use mocking frameworks to create dummy mock objects. *Stub* related methods are also among the most popular API—this indicates that developers frequently use mocking frameworks to control the method behavior of the mock objects. *Verify* is another popular API for checking method execution status of the mock objects.

5.3. RQ3: Mocked Dependencies

Table 8 shows the number of mocked library dependencies, as well as the percentage of mocked library dependencies in all mocked dependencies. On average, the majority (61.1%) of the mocked objects are for isolating dependencies to external libraries. In some projects, this percentage reaches 100%. This indicates that mocking is an important way to isolate dependencies to external libraries.

Furthermore, Table 9 lists the most frequently mocked external libraries. The results show that the most frequently mocked library classes are related to Modularization System, HTTP request/response, Database, and IO/File System. We imply that a key motivation for Apache developers to use a mocking framework is to 1) Prevent interference from external libraries (such like the concurrent nature of OSGI framework) and 2) improve the testing performance through avoiding accessing a real HTTP server, database, or file system.

Table 9
Most Frequently Mocked Library Classes

Mocked Library Class	Type	Frequency
org.osgi.framework.Bundle	modularization System	112
javax.servlet.http.HttpServletRequest	HTTP Request/Response	70
javax.servlet.http.HttpServletResponse	HTTP Request/Response	56
org.apache.hadoop.fs.FileSystem	IO/File System	39
org.apache.hc.core5.http.HttpConnection	HTTP Request/Response	38
org.osgi.framework.BundleContext	modularization System	37
java.io.File	IO/File System	35
org.slf4j.Logger	Logging System	35
java.sql.Connection	Database	32
java.sql.ResultSet	Database	32

Table 10
Mock Files without Mocking Frameworks Dependencies

	Avg.	Med.	Max.	Min.
#Mock Files without Mocking Frameworks Dependencies	3.04	0	158	0
Proportion of Mock Files without Mocking Frameworks Dependencies	0.6%	0%	11.5%	0%

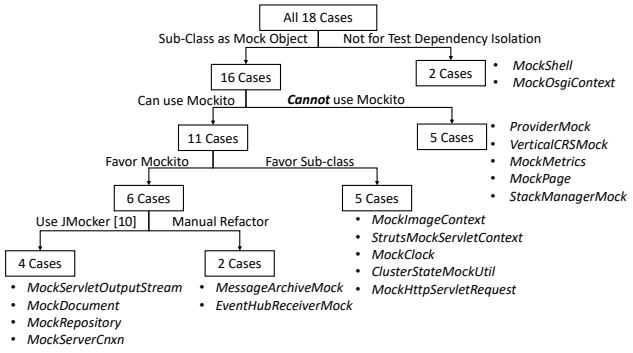
Summary: Developers from Apache projects tend to mock more library classes (61.1%) than the classes in their own package (38.9%). The most frequently mocked library API is HTTP request/response, it indicates that the common reason for mocking the library classes is to prevent calling the real HTTP services and improve testing performance.

5.4. RQ4: Mocks without Leveraging Mocking Frameworks

Table 10 shows that, there are on average 3 files—and up to 158 (11.5%) files—in a project that associate with the concept of "mocking" without relying on any mocking framework.

We randomly sampled 18 such cases to investigate how and why developers use the concept of mocking without a mocking framework. Figure 5 shows the classification of these cases.

Figure 5: Classification of the 18 Sample Cases



In 16 cases (the left child of the root node in Figure 5), the developers use sub-classes as mock objects. That is, the

Electronic copy available at: https://ssrn.com/abstract=4100265

developers create a sub-class and leverage method overriding to control the behavior of the sub-class for mocking purposes. For example, *MockServletOutputStream*[1] is a sub-class of *ServletOutputStream*. The former mocks the latter, and alters the latter's behavior by overriding the method *write* for testing purpose.

Among these 16 cases that use inheritance for mocking, 11 cases can potentially use a mocking framework, such as *Mockito*, to replace the sub-classes. However, only in 6 cases, using *Mockito* may be favored over sub-classing to improve the quality of test cases. While in the other 5 cases, using inheritance for mocking may be favored, since it is more convenient and tangible than using a mocking framework, especially for developers who do not have in-depth experience with a mocking framework.

As revealed in our prior work [10], using *Mockito* to replace inheritance for mocking could make the test logic more clear, easier to understand and maintain. For the 6 cases that favor *Mockito*, we found that 4 cases can be refactored automatically by our tool, JMocker [10], to replace the inheritance by using *Mockito*. These 4 cases are *MockServletOutputStream*, *MockDocument* [2], *MockRepository* [3], and *MockServerCnxn* [4]. The other 2 cases, *MessageArchivesMock* [5] and *EventHubReceiverMock* [6], can be refactored with some manual effort based on our experience.

Among the other 5 cases that favor inheritance for mocking, one case is a concrete-class, *MockImageContext* [7], which implements *ImageContext*. It is designed as a Singleton. The creation of a Singleton mock object using a mocking framework is not as straight-forward as using inheritance. In 2 cases— *StrutsMockServletContext*[8] and *MockClock*[9]—the sub-classes have a large number of references in test cases. For example, sub-class *StrutsMockServletContext* is used in 8 different test cases. Mock objects created using a mocking framework are used within the scope of a test case or a test class, where the mock objects are created. In comparison, sub-class provides better reusability since it is accessible in the scope of the entire project. In another 2 cases—*ClusterStateMockUtil*[10] and *MockHttpServletRequest*[11]—the sub-classes contain quite complicated logic, with many attributes and overriding methods. Although creating such mock objects using a mocking framework may still be feasible, it is not as straight-forward and convenient as using sub-classes.

In the remaining 5 cases of the 16 cases, it is not feasible to replace inheritance by using a mocking framework due to the limitations of existing mocking frameworks. More specifically, 1) the sub-classes use special modifiers, including *ProviderMock* [12] and *VerticalCRSMock* [13]. They both use modifier *strictfp*—which a mocking framework cannot handle. 2) Two sub-classes inherit and implement multiple classes, including *MockMetrics* [14] and *MockPage* [15]. This indicates mocking multiple classes, which is not supported by existing mocking frameworks. For example, *MockPage* extends *MockComponent* and implements *IRequestablePage* at the same time. And, 3) the sub-class contains static fields, and also alters the behavior of equals() and hashCode(), which cannot be handled using a mocking framework. This case is *StackManagerMock*[16]

In the last 2 cases (in the right child of the root node in Figure 5)—*MockShell* [17] and *MockOsgiContext* [18]—the concept of mocking is *not* intended to address test dependency isolation as intended by the mocking frameworks. More specifically, the class *MockShell* from project, *Accumulo*, aggregates the class *Shell* as an attribute. The *MockShell* add more functions to *Shell* for testing purposes, such

[1] MockServletOutputStream https://github.com/apache/shindig/blob/8f3c3d5c77f5324bad56a5a62da28657fe9112a0/java/social-api/src/test/java/org/apache/shindig/social/core/oauth/MockServletOutputStream.java

[2] MockDocument https://github.com/apache/creadur-rat/blob/008161c5b5c4119fc911004709cb089e11d974d8/apache-rat-core/src/test/java/org/apache/rat/document/MockDocument.java

[3] MockRepository https://github.com/apache/ace/blob/3f878e4e99c003a92f20f4698b40bba5b719f67b/org.apache.ace.client.repository/test/org/apache/ace/client/repository/impl/MockRepository.java

[4] MockServerCnxn https://github.com/apache/zookeeper/blob/c74658d398cdc1d207aa296cb6e20de00faec03e/zookeeper-server/src/test/java/org/apache/zookeeper/server/MockServerCnxn.java

[5] MessageArchivesMock https://github.com/apache/mina-vysper/blob/master/server/extensions/xep0313-mam/src/test/java/org/apache/vysper/xmpp/modules/extension/xep0313_mam/spi/MessageArchivesMock.java

[6] EventHubReceiverMock https://github.com/apache/storm/blob/3f96c249cbc17ce062491bfbb39d484e241ab168/external/storm-eventhubs/src/test/java/org/apache/storm/eventhubs/spout/EventHubReceiverMock.java

[7] MockImageContext https://github.com/apache/xmlgraphics-commons/blob/e817d9d5a3fe3b1b56fd9e6e288b3da90bdf9e60/src/test/java/org/apache/xmlgraphics/image/loader/MockImageContext.java

[8] StrutsMockServletContext https://github.com/apache/struts/blob/49a4d6d5a11227314a5412935b31989fad3bffc9/core/src/test/java/org/apache/struts2/views/jsp/StrutsMockServletContext.java

[9] MockClock https://github.com/apache/tez/blob/2dcbe0bca22be89a797acd5f2228d91c4c112069/tez-dag/src/test/java/org/apache/tez/dag/app/MockClock.java

[10] ClusterStateMockUtil https://github.com/apache/solr/blob/73e64a959c93f721fd72e9d701cd8f3d925c9688/solr/core/src/test/org/apache/solr/cloud/ClusterStateMockUtil.java

[11] MockHttpServletRequest https://github.com/apache/wookie/blob/b75542fc42ea081aaa8947549e368c31a797eaa4/wookie-server/src/test/java/org/apache/wookie/tests/helpers/MockHttpServletRequest.java

[12] ProviderMock https://github.com/apache/sis/blob/700b4574ff46a229b4a336bbab7ebf3e6b4f0093/core/sis-referencing/src/test/java/org/apache/sis/internal/referencing/provider/ProviderMock.java

[13] VerticalCRSMock https://github.com/apache/sis/blob/5dbfe58a0fb7a804354db6c410280eb0a84288a1/core/sis-metadata/src/test/java/org/apache/sis/test/mock/VerticalCRSMock.java

[14] MockMetrics https://github.com/apache/skywalking/blob/f5b7c3e32d022973050340e64dc63835787f5ad8/oap-server/exporter/src/test/java/org/apache/skywalking/oap/server/exporter/provider/grpc/MockMetrics.java

[15] MockPage https://github.com/apache/wicket/blob/837f3c137bf39f26ddc3b8e939235cde04e8c13d/wicket-core/src/test/java/org/apache/wicket/MockPage.java

[16] StackManagerMock https://github.com/apache/ambari/blob/24dbed27b97f4c82835273758f0bceb3334b4ef2/ambari-server/src/test/java/org/apache/ambari/server/stack/StackManagerMock.java

[17] MockShell https://github.com/apache/accumulo/blob/8b8a6f7339fdcfbf4118c6164eb47035d857f0d9/test/src/main/java/org/apache/accumulo/test/shell/MockShell.java

[18] MockOsgiContext https://github.com/apache/stanbol/blob/2fcf471b467fb84e59491f4c54ba0a95924ab04a/ontologymanager/registry/src/test/java/org/apache/stanbol/ontologymanager/registry/MockOsgiContext.java

Electronic copy available at: https://ssrn.com/abstract=4100265

as adding a set of *assertion* statements. The *MockShell* is involved in integration test cases, such as *ShellConfigIT*, to replace the original *Shell*, for the convenience of testing. In the other case, *MockOsgiContext* from project, *Stanbol*, simply provides a function called *reset* to reset a variable called *TcManager*. This case is not mocking any other class in the project. Thus, the concept of "mocking" is not always used as intended for test dependency isolation.

Summary: There are up to 11.5% test files used as mock object without relying on any mocking frameworks. Some of these cases can be replaced by using a mocking framework; while others point to the potential limitations with existing mocking frameworks.

5.5. Comparison with Mostafa and Wang's Study

This section compares our study results with that of the Mostafa and Wang's study:

Dataset Comparison: Mostafa and Wang's study focused on 5000 random selected open source projects in GitHub. Many of these projects are relatively small-scale, and individually owned. In comparison, our study focuses on 193 open source projects from Apache, many of which are large-scale, popular in different problem domains, such as Hadoop and Cassandra. Overall, the most distinctive feature of our dataset is that the projects are in larger scale: The average size of our study subjects is 164K LOC, comparing to 25K LOC of the original study. We assume that project scale could be an important factor that impacts whether and how projects use mocking frameworks. Overall, we imply that more complex projects are in greater need for the sophisticated usage of a mocking framework. Following, we make comparison between our study and Mostafa and Wang's study based on the three RQs.

Popularity of Mocking Frameworks: In Mostafa and Wang's study [11], about 23% projects use mocking frameworks. In comparison, 67% of the Apache projects in our study use mocking frameworks. We believe that this higher adoption of mocking frameworks in our dataset is determined by the larger project scale—projects in our dataset, on average, are more than six times the size of the projects in the prior study. This confirms our conjecture that larger-scale and more complicated projects are in greater need for mocking frameworks.

It is also worth noting that 12% projects use springframework [27], which is not found in Mostafa and Wang's study [11]. We find these 16 projects all use web services such as Wink [28], a Java based framework that provides functionality for communicating with RESTful Web services and Struts [29], an open source MVC frameworks for java web application. This indicates that web application services tend to use springframework [27] as their mocking framework. This suggests that the type of mocking frameworks used by projects relates to the functionality of the projects.

Usage of Mocking Frameworks: In RQ2, both our empirical study and Mostafa and Wang's study [11] analyzed the top 10 most popular APIs in EasyMock and Mockito. Both empirical studies show that the most frequently used mocking APIs are related to method stubbing, mock creation and behavior verification. This indicates that testers are not use mocking frameworks to simply create dummy objects, they frequently leverage the advanced mechanisms to control the behavior of the mock objects as well as verifying the interactions between mock objects and other production classes.

Mocked Dependencies: In our dataset, the majority (61.1%) of the mocked objects are for replacing library classes. In comparison, in the prior study, a smaller portion (39.4%) of the mocked objects are for library classes. We believe that this is relevant to the project domains. Our study subjects contain many web applications such like Wink [28] and Struts [29], which use springframework for mocking web services. For example, Struts [29] is an open source framework for creating Java web applications. Developers use springframework to mock HttpServletRequest and HttpServletResponse as input and the expected outputs to test various RESTful APIs. For these web application projects, developers leverage springmock to prevent the interference of unstable network and improve run-time testing performance.

5.6. Study Implications

Mocking frameworks are widely used in real-life projects, i.e. in 67% of Apache projects and in 23% of GitHub projects. This indicates that it is important for software practitioners to learn and get familiarized with how to use mocking frameworks to overcome the challenge of test dependency isolation in unit testing. In particular, Mockito and EasyMock are the most popular mocking frameworks. Thus, it is most beneficial for software engineering educators to develop related curriculum materials regarding the usage of mocking frameworks based off Mockito and EasyMock.

As suggested in our study, the adoption of mocking frameworks in a software project may be impacted by the project's characteristics, such as project size and problem domain. For example, large projects are more likely to in need of a mocking framework than a small project. In addition, projects related to web-applications tend to use SpringFramework for mocking HTTP requests/responses to facilitate unit testing. Our study shed light on the factors, such as project size and domain, that may impact the adoption of a mocking framework. However, more in-depth and quantitative investigation still requires future research to guide practitioners in the adoption of a proper mocking framework for their projects.

In both Mockito and EasyMock, we identified most frequently used APIs. For example, in Mockito, *mock* is most frequently used for creating mock objects, *when* is most frequently used for method stubbing, and *verify* is frequently used for verifying mock object behavior. Practitioners can benefit from the common API sequences for the mock object

Electronic copy available at: https://ssrn.com/abstract=4100265

creation, manipulation, and verification. It is still open to future research to extract the most common API sequences to facilitate the learning and usage of popular mocking frameworks.

As suggested in the RQ4, sub-classing seems to be a common mechanism for mocking without relying on a mocking framework. In some cases, the sub-classing could be easily replaced by using a mocking framework to benefit the quality and maintainability of test cases. Automated refactoring tools to replace sub-classing by a mocking framework, especially based on Mockito or EasyMock, could provide great value for practitioners to facilitate the usage of mocking frameworks.

While, in some cases, sub-classing for mocking seems to be more convenient and tangible than using a mocking framework, especially for developers who lacks in-depth experience of using a mocking framework. For example, creating a sub-class as a mock object can be easily reused in different test cases. While, it requires advanced design to create a reusable mock object using a mocking framework. That is, developers could create a dedicated class, which aggregates a mock object created using a mocking framework. As such the aggregating class can be reused in different test cases in the same way as a sub-class. However, there is currently little empirical knowledge regarding the advanced design solutions for mock objects using mocking frameworks.

Finally, in some cases, sub-classing cannot be replaced by using a mocking framework, due to the limitations of existing frameworks. It is open to future research to systematically investigate the limitations of existing mocking frameworks. Practitioners should be aware of related scenarios and limitations with existing mocking frameworks. Also, designers and developers of existing mocking frameworks should think of ways to address the limitations.

6. Limitations and Threat to Validity

We cannot guarantee that the scripts we created for extracting and analyzing the usage of mocking frameworks are free of bugs. To mitigate this threat, we conducted manual verification of the experiment results based on sampled projects in each of the study steps. We were able to identify and fix several minor issues in the scripts that lead to inaccuracy of our results. Thus, we believe that the study results presented in this paper are reliable. We have publicized our scripts and intermediate data here `https://github.com/RedRoach51/MockResearch`.

It is a limitation that we were not able to analyze 16 projects due to issues in the project configuration. As mentioned earlier, our analysis scripts are based on Eclipse JDT libraries. We were not able to successfully import these projects to proceed with our analysis. We admit that our study results would be more comprehensive if these projects were successfully analyzed. However, we believe that this would not affect the overall findings of this study, since the 193 projects are already great representation.

Another limitation is that this study only focuses on projects implemented in Java. Therefore, the mocking frameworks and their APIs are also based on Java. We cannot guarantee that similar results would hold for projects implemented in a different programming language. There are frameworks that dedicated to other languages such as Python and JavaScript, which are out of the scope of this study. We believe that the programming language may have an impact on the convention of how mocking is done. We plan to explore this further in future studies.

Our analysis is based on the current code base of the Apache java projects. Since Apache is a quite active community, their projects are under-going continuous changes. That means, if, in the future, other researchers try to replicate our study, we cannot guarantee that the same conclusions will be found. In addition, this study does not reveal the evolution of how mocking frameworks are being used in a project between when it is initially created to when it is developed in a more stable stage. Our hypothesis is that the usage of mocking frameworks may be impacted by the evolution and maturity of the projects. We think that this is a worthwhile hypothesis to be evaluated in future research.

Finally, we cannot guarantee that the same observations will hold for projects from a different platform. As we have shown in Section 5.5, how developers use mocking frameworks may be impacted by project characteristics, such as size and domain. If the study is replicated on a different dataset, the observations may vary depending on the characteristics of the projects.

7. Related Work

In the past decade, research related to mocking in software testing has drawn increasing interests. Freeman et al.[30] was one of the first to propose the basic idea of mocking in unit testing. They contributed a mocking framework named jMock for Java [31].

In following years, researchers start to expand the usage of mocking in the unit testing of new domains, such as embedded systems, cloud computing, and mobile applications. Karlesky et al.[32] introduced mocking in testing embedded software systems. They proposed a holistic set of practices, tools, and a new design pattern to apply the Test-Driven Development with mocking frameworks in embedded software systems. Their methodology can reduce the software flaws and improve the progress in data-driven project management in embedding software development. Kim et al.[33] explored the challenges of mocking framework in the unit testing of embedded systems as well. The study pointed out that embedded software was tightly coupled with target hardware. They showed how mocking frameworks could help to improve the design process, the architecture of the software components, and protect the system against regression defects. Svensgård et al.[34] proposed the idea of using mocking frameworks in testing SaaS cloud platform. The study leverages mock objects for replacing the dependency to cloud data instance in unit testing. The study showed

Electronic copy available at: https://ssrn.com/abstract=4100265

that testing based on mocking can find the same faults as testing against the real cloud, and at the same time keep the same code coverage. Fazzini et al.[35] proposed an improved mocking framework MOKA, which is specialised in generating reusable mock objects for mobile apps unit testing. It uses component-based program synthesis to leverage existing test executions to generate mock objects automatically. The study shows that this helps developers to repair the tests that have external data dependency.

With the prevalent usage of mocking framework, researchers also started to focus on improving the education and design of mock objects. Nandigam et al.[36] shared the teaching experience of implementing mock objects in a interface-based system. The study showed that implementing mock objects can helps students to test their system as units in isolation and develop code that adheres to the critical principles of reusable object-oriented objects. Solms et al.[37] proposed a contract-based design to reuse the mock objects in the services-oriented development. Mock objects were tested against the component contracts, which improved the re-usability of the mock objects in both the unit test and integration test. However, this also required more code to be developed for specifying mocking behavior. Pereira et al.[38] investigated the design of hand-coded mocking objects in modern projects. The study pointed out the over creation of private mock classes is widespread. Marri et al.[22] investigated the benefits of using mock objects when testing the file-system-dependent software. The study identified two benefits: mock objects enable unit testing of the code that interacts with external APIs, and improve the code coverage in unit testing.

In recent years, automated tools for enforcing the usage of mocking frameworks started to emerge. Arcuri et al.[39] incorporated a mocking framework to automated unit test generation. Their study confirmed the anticipated improvements in code coverage and bug detection. Zhu et al.[40] introduced a new machine learning based tool to identify and recommend mocks for unit tests. The tool requires only the class under test and the class's dependency information as the input. It outperformed three baseline approaches: existing heuristics, EvoSuite mock list, and Empirical rules. Wang et al. [10] contributed an approach to automatically identify and refactor the test cases using inheritance with mock objects using Mockito. The refactoring tool reduced code complexity, provided efficient run-time performance in real-life projects, and was applicable to general datasets.

As mentioned earlier, Mostafa et al.[11] was the first to study the usage of the mocking frameworks in practice. They conducted an empirical study on more than 5000 open source projects from Github, analyzing the quantity of projects using mocking framework and which Mocking framework was most used. Their result shows that 23% of the projects use at least one mocking framework, and Mockito is the most popular mocking framework. Spadini et al.[7, 8] investigated the usage and evolving process of mock objects in three OSS projects and one industrial system. The study proposed a mocking data mining tool MOCKEXTRACTOR. They also conducted a structured survey with more than 100 professionals. The result revealed that developers frequently mock dependencies that make testing difficult and prefer not to mock classes that encapsulate domain concepts/rules of the system. This indicates that the usage of mock objects could depend on the responsibility and the architectural concern of the classes.

8. Conclusion

Mocking frameworks are used in 129 (67%) out of the 193 Apache projects. We found that larger projects are more likely to use a mocking framework—the median size of projects with a mocking framework is 122k LOC compared to the 77k LOC for those without a mocking framework. Mockito, EasyMock and PowerMock are the top three most popular mocking frameworks used in 71%, 27%, and 12% projects. Developers leverage advanced mocking APIs, such as *when*, *expect*, and *verify* in Mockito, to manipulate and verifying the behavior of mock objects for the purpose of testing. In Apache projects, the majority (61.1%) of the mocked objects are for isolating dependencies to external library classes rather than classes that are internal to the projects. Finally, up to 11.5% files in a project associate with the concept of "mocking" without relying on any mocking framework. Some of these cases can be improved by using a mocking framework, while the others cases contain complicated mocking logic that are currently not supported by existing mocking frameworks. Practitioners who are interested in adopting mocking frameworks can gain insights regarding when and how mocking frameworks are used in Apache projects.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation (NSF) under grants CCF-1909085 and CCF-1909763.

References

[1] P. Runeson, A survey of unit testing practices, IEEE software 23 (2006) 22–29.

[2] Ieee standard glossary of software engineering terminology, IEEE Std 610.12-1990 (1990) 1–84.

[3] C. Kaner, J. Falk, H. Q. Nguyen, Testing computer software, John Wiley & Sons, 1999.

[4] E. Daka, G. Fraser, A survey on unit testing practices and problems, in: 2014 IEEE 25th International Symposium on Software Reliability Engineering, IEEE, 2014, pp. 201–211. doi:10.1109/ISSRE.2014.11.

[5] V. Garousi, J. Zhi, A survey of software testing practices in canada, Journal of Systems and Software 86 (2013) 1354–1376.

[6] A. Bertolino, Software testing research: Achievements, challenges, dreams, in: Future of Software Engineering (FOSE'07), IEEE, 2007, pp. 85–103. doi:10.1109/FOSE.2007.25.

[7] D. Spadini, M. Aniche, M. Bruntink, A. Bacchelli, To mock or not to mock? an empirical study on mocking practices, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 402–412. doi:10.1109/MSR.2017.61.

Electronic copy available at: https://ssrn.com/abstract=4100265

[8] D. Spadini, M. Aniche, M. Bruntink, A. Bacchelli, Mock objects for testing java systems, Empirical Software Engineering 24 (2019) 1461–1498.

[9] K. Taneja, Y. Zhang, T. Xie, Moda: Automated test generation for database applications via mock objects, in: Proceedings of the IEEE/ACM international conference on Automated software engineering, 2010, pp. 289–292.

[10] X. Wang, L. Xiao, T. Yu, A. Woepse, S. Wong, An automatic refactoring framework for replacing test-production inheritance by mocking mechanism, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 540–552.

[11] S. Mostafa, X. Wang, An empirical study on the usage of mocking frameworks in software testing, in: 2014 14th international conference on quality software, IEEE, 2014, pp. 127–132. doi:10.1109/QSIC.2014.19.

[12] https://www.apache.org/, .

[13] https://site.mockito.org/, .

[14] https://easymock.org/, ????

[15] https://powermock.github.io/, ????

[16] https://mvnrepository.com/artifact/org.springframework/spring-mock, .

[17] https://docs.python.org/3/library/unittest.mock.html#module-unittest.mock, ????

[18] http://nmock.sourceforge.net/, ????

[19] https://github.com/moq/moq4, ????

[20] F. Henderson, Software engineering at google, arXiv preprint arXiv:1702.01715 (2017).

[21] A. Hunt, D. Thomas, Pragmatic unit testing in c# with nunit, The Pragmatic Programmers, 2004.

[22] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, W. Schulte, An empirical study of testing file-system-dependent software with mock objects, in: 2009 ICSE Workshop on Automation of Software Test, IEEE, 2009, pp. 149–153. doi:https://doi.org/10.1007/s10664-018-9663-0.

[23] https://projects.eclipse.org/projects/eclipse.jdt, .

[24] https://junit.org/junit5/, ????

[25] https://pdfbox.apache.org/, ????

[26] X. Wang, Understanding and Facilitating the Usage of Mocking Frameworks for Test Dependency Isolation, Ph.D. thesis, Stevens Institute of Technology, 2021.

[27] https://spring.io/, .

[28] http://wink.apache.org/, .

[29] https://struts.apache.org/, .

[30] S. Freeman, T. Mackinnon, N. Pryce, J. Walnes, Mock roles, not objects, in: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2004, pp. 236–246. doi:https://doi.org/10.1145/1028664.1028765.

[31] S. Freeman, T. Mackinnon, N. Pryce, J. Walnes, jmock: supporting responsibility-based design with mock objects, in: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2004, pp. 4–5.

[32] M. Karlesky, G. Williams, W. Bereza, M. Fletcher, Mocking the embedded world: Test-driven development, continuous integration, and design patterns, in: Proc. Emb. Systems Conf, CA, USA, 2007, pp. 1518–1532.

[33] S. S. Kim, Mocking embedded hardware for software validation, Ph.D. thesis, 2016.

[34] S. Svensgård, J. Henriksson, Mocking saas cloud for testing, 2017.

[35] M. Fazzini, A. Gorla, A. Orso, A framework for automated test mocking of mobile apps, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2020, pp. 1204–1208.

[36] J. Nandigam, V. N. Gudivada, A. Hamou-Lhadj, Y. Tao, Interface-based object-oriented design with mock objects, in: 2009 Sixth International Conference on Information Technology: New Generations, IEEE, 2009, pp. 713–718. doi:10.1109/ITNG.2009.268.

[37] F. Solms, L. Marshall, Contract-based mocking for services-oriented development, in: Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, 2016, pp. 1–8.

[38] G. Pereira, A. Hora, Assessing mock classes: An empirical study, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 453–463. doi:10.1109/ICSME46990.2020.00050.

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

[40] H. Zhu, L. Wei, M. Wen, Y. Liu, S.-C. Cheung, Q. Sheng, C. Zhou, Mocksniffer: Characterizing and recommending mocking decisions for unit tests, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 436–447.

Electronic copy available at: https://ssrn.com/abstract=4100265