Empirical Software Engineering (2024) 29:39
https://doi.org/10.1007/s10664-023-10410-y

An empirical study on the usage of mocking frameworks in Apache software foundation

Lu Xiao[1] · Gengwu Zhao[1] · Xiao Wang[1] · Keye Li[1] · Erick Lim[1] · Chenhao Wei[1] · Tingting Yu[2] · Xiaoyin Wang[3]

Accepted: 10 October 2023 / Published online: 23 January 2024
© The Author(s) 2024

Abstract
Mocking frameworks provide convenient APIs, which create mock objects, manipulate their behavior, and verify their execution, for the purpose of isolating test dependencies in unit testing. This study contributes an in-depth empirical study of whether and how mocking frameworks are used in Apache projects. The key findings and insights of this study include: First, mocking frameworks are widely used in 66% of Apache Java projects, with Mockito, EasyMock, and PowerMock being the top three most popular frameworks. Larger-scale and more recent projects tend to observe a stronger need to use mocking frameworks. This underscores the importance of mocking in practice and related future research. Second, mocking is overall practiced quite selectively in software projects—not all test files use mocking, nor all dependencies of a test target are mocked. It calls for more future research to gain a more systematic understanding of when and what to mock to provide formal guidance to practitioners. On top of this, the intensity of mocking in different projects shows different trends in the projects' evolution history—implying the compound effects of various factors, such as the pace of a project's growth, the available resources, time pressure, and priority, etc. This points to an important future research direction in facilitating best mocking practices in software evolution. Furthermore, we revealed the most frequently used APIs in the three most popular frameworks, organized based on the function types. The top five APIs in each functional type of the three mocking frameworks usually take the majority (78% to 100%) of usage in Apache projects. This indicates that developers can focus on these APIs to quickly learn the common usage of these mocking frameworks. We further investigated informal methods of mocking, which do not rely on any mocking framework. These informal mocking methods point to potential sub-optimal mocking practices that could be improved, as well as limitations of existing mocking frameworks. Finally, we conducted a developer survey to collect additional insights regarding the above analysis based on their experience, which complements our analysis based on repository mining. Overall, this study offers practitioners profound empirical knowledge of how mocking frameworks are used in practice and sheds light on future research directions to enhancing mocking in practice.

Communicated by: Dan Hao

✉ Lu Xiao
lxiao6@stevens.edu

Extended author information available on the last page of the article

Keywords Software testing · Mocking frameworks · Apache open-source projects

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 (Runeson 2006; Ieee standard 1990; Kaner et al. 1999; Daka and Fraser 2014; Garousi and Zhi 2013). A unique challenge in unit testing results from the inter-dependencies among the units (Runeson 2006; Bertolino 2007). 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 (Spadini et al. 2017, 2019). 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 (Taneja et al. 2010). 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 like 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 (Wang et al. 2021) 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 (2014) conducted an empirical study on how mocking frameworks are used in the GitHub community. Their study focused on 5000 software projects from Git. Their study contributed several key findings: 459 (23%) projects, which contain test code, uses at least one mocking framework. Only a small portion (17%) of all dependency classes are mocked in the projects. The top four

most popular frameworks used by projects on GitHub include, Mockito, EasyMock, JMock, and JMockit. 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. Software testers tend to use 60% of the mock objects for replacing source code classes and the remaining 40% for library classes.

At a high level, this study is motivated to investigate the usage of mocking frameworks in Apache projects since Apache Software Foundation has been widely recognized and researched as a distinguished example of successful open-source software communities (Mockus et al. 2002, 2000; Crowston and Howison 2006). Practitioners and researchers have been gaining experience and insights from the successful practices of Apache projects to lead the open-source movement (Rigby et al. 2008; Duenas et al. 2007; Weiss et al. 2006). In addition, Apache provides a meeting point where engineers from large companies like IBM, Google, Yahoo, Sun, and Oracle work as volunteers to build open-source software infrastructure (Severance 2012). No prior studies have revealed how mocking is practiced in Apache projects. Thus, we are motivated to fill this gap by providing empirical insights regarding what project factors impact the adoption of mocking frameworks, as well as the specifics of how mocking frameworks are adopted by Apache projects.

In this study, we specifically focus on the 246 Apache Java projects (Apache software foundation projects list 2023), since Java has been one of the most widely used programming languages in the past decades. To enable automated analysis of how mocking frameworks are used in the hundreds of Java projects, many of which are complicated and large-scale, we created a source code parser to search for the usage of related mocking framework APIs based on the Eclipse JDT. This parser is language-dependent, and cannot be used directly for another language, such as Python. The analysis of how mocking frameworks are being used in other languages is also a valuable future direction but is out of the scope of this study.

This study contributes important empirical knowledge regarding the mocking practice, by focusing on five complementary RQs that have not been sufficiently addressed in related prior studies. More specifically, we summarize the key findings and insights of this study based on the five RQ1. In RQ1, we investigated the overall adoption of mocking frameworks in Apache projects. We found that 66% of Apache projects use at least one mocking framework, with Mockit, EasyMock, and PowerMocknig being the most popular. In particular, larger-scale and newer projects are more like to use a mocking framework. This highlights the overall importance of mocking in practice and its related research, especially for modern, large-scale, and complicated projects. In RQ2, we investigated the intensity of mocking across different projects. We found that mocking is practiced quite selectively in a project in general. This points to important future research in providing formal guidance to facilitate mocking practice, regarding when to mock, what to mock, and who should be in charge of mocking. In RQ3, we examined the trends of mocking practice over the projects' evolution history. We found that the usage intensity of mocking in different projects shows different trends, and the number of developers who worked on mocking was quite dynamic, in the evolution history. The implication is that there are compound effects of various factors, such as the pace of a project's growth, available resources, time pressure, and project priority that impact the dynamism of mocking in projects' evolution. It calls for more future research to understand these compound factors to facilitate mocking practice in the context of software evolution. In RQ4, we revealed the most popular mocking function APIs from the three most popular frameworks. This benefits practitioners with a "cheat sheet" to accelerate their learning curve in using mocking frameworks. In RQ5, we discovered informal mocking methods that do not rely on any mocking frameworks. These cases point to sub-optimal mocking practice, as well as potential limitations of existing mocking frameworks. This RQ also sheds light on the

future research direction regarding how to eliminate sub-optimal mocking and address the limitations of existing frameworks. Finally, we complemented the repository-mining-based analysis of the RQs through a survey participated by 17 Apache developers. The survey confirmed our findings and also provided additional insights from real-world developers' experience.

Despite numerous prior studies that focus on mocking, which are discussed in detail in Section 9, the most relevant study to ours is by Mostafa and Wang (Mostafa and Wang 2014), which provides empirical experience regarding mocking frameworks in the GitHub community. When comparing to Mostafa and Wang (2014), this study distinguishes itself in the two aspects:

- Our study provides complementary knowledge to Mostafa and Wang's study by focusing on projects with different "demographics" features. the Apache software foundation hosts many widely adopted, long-lived open-source projects, such as Cassandra, Tomcat, Hadoop, etc. Not only are these projects contributed by large, geographically distributed teams, but also they serve as the foundation for various commercial software systems that greatly benefit society. In comparison, GitHub hosts smaller and individual-owned projects. As highlighted in Section 6, we made a detailed, statistical comparison of the "demographics" of projects on Apache and on GitHub, regarding the team size, activity level, project history, and project scale. The findings underscore the different project characteristics on these two platforms. Our assumption is that whether and how mocking frameworks are adopted in these two communities could be impacted by the project characteristics.
- The depth of this study is more significant in all the RQs compared to Mostafa and Wang (2014), although our study only contains 193 Apache projects, while Mostafa and Wang (2014) examined 5000 Github projects. More specifically, in RQ1, we conducted a statistical analysis of what project factors may impact mocking framework adoption, which is not available in Mostafa and Wang (2014). In RQ2, we provided a more detailed statistical analysis of the intensity of mocking framework adoption from three measures—the percentage of test files that use mock, the percentage of mocked dependencies, and the distribution of mock objects and mocked dependencies in files. In RQ3, we conducted a thorough and quantitative investigation of how the mocking framework adoption evolved over the projects' history, which is not available in Mostafa and Wang (2014). In RQ4, we conducted a systematic analysis of mocking framework APIs based on their functions and revealed the top five frequently used APIs. In comparison, Mostafa and Wang (2014) only presented the top 10 APIs in each framework without qualitatively differentiating different functions. In RQ5, we also investigate how practitioners may use informal methods, instead of a mocking framework, for mocking practice. For example, creating a sub-class as a mock object is common, which is commonly referred to as "stub" and "fake" in the grey literature. Mostafa and Wang (2014) did not focus on these problems. Finally, we conducted a survey to collect input from developers to confirm and deepen our understanding of the RQs to better illuminate future research. This is also not available in Mostafa and Wang (2014).

The rest of this paper is organized as follows. Section 2 discusses background information. Section 3 presents the research questions we aim to address in this study. Section 4 introduces our study process. Section 5 elaborates our findings. Section 6 compares our findings with that of Mostafa and Wang's study. Section 7 discusses the implications of this study and future directions. Section 8 discusses limitations and threats to validity. Section 9 discusses related work. Section 10 concludes this study.

2 Background

2.1 Motivating Example for Mocking

Figure 1a 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 sends requests to and receives 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 Fig. 1b, 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 return true to indicate the request was successfully received and processed or return false to test the fault tolerant mechanism for the customer services. 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, such as Mockito (mockito 2023), EasyMock (EasyMock 2023), PowerMock (PowerMock framework site 2023) and SpringframeworkMock (Maven Repository 2023) for Java; Mock (Unittest 2023) for Python; NMock (NMock 2023), Moq (moq 2023) for C#. These frameworks are widely

(a) Test Dependencies on Web Server and SQL Database (b) Dependency Isolation by an Mock Object

Fig. 1 : Comparing Mock and Real Dependencies

used in software projects to ease the process of unit testing (Henderson 2017; Hunt and Thomas 2004; Marri et al. 2009).

Figure 2 is the implementation of the motivating example in the previous subsection, implemented following the syntax of Mockito. The function under test (FUT) is the *subscribeCustomer* method (called in line 31) in the *CustomerService* class. 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). That is, whenever *emailManager.subscribe()* is called, we invoke *emailManager.sendEmail()* which always return *true* through the lambda expression. 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).

Note that despite the advantages of mocking frameworks, not all developers fully support their use. For instance, Spadini's study (Spadini et al. 2017, 2019) points out that using mocks can bring several challenges such as maintaining the compatibility of the mock's behavior with the original class, the relationship between the amount of mocking required for a test class and its code quality, and the (unfavorable) excessive use of mock objects to test legacy systems. In communities like StackOverflow (Why is it so bad to mock classes? 2023), developers often debate the use of mocks. Some argue that while stubs are often desirable, they are overused through mocking frameworks, resulting in fragile tests. They find striking the right balance challenging. Others emphasize programming to interfaces over implementations, viewing frequent class mocking as a misuse. They also warn that excessive mocking necessitates making virtually everything mockable, potentially disrupting the class design. Our research question RQ5 also examines this phenomenon and has found instances of implementing mocks through Customized Mock Classes rather than using mocking frameworks. We also got related confirmation from the survey we designed for this study.

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?* The goal of this RQ is to reveal the overall popularity of the mocking frameworks among all the Apache projects. That is, what is the percentage of Apache projects that adopt a mocking framework? In addition, we analyze this RQ in-depth in two parts:

```
23 class TestCustomerService {
24     @Test
25     public void testSubscribeCustomer() {
26         EmailManager emailManager = mock(EmailManager.class);
27         Mockito.when(emailManager.subscribe()).thenAnswer(invo -> {
28             emailManager.sendEmail();
29             return true;});
30         CustomerService myservice = new CustomerService();
31         myservice.subscribeCustomer(emailManager);
32         myservice.emailCustomers(emailManager);
33         Mockito.verify(emailManager, Mockito.atLeastOnce()).subscribe();
34         Mockito.verify(emailManager, Mockito.times(2)).sendEmail();}
35 }
```


Fig. 2 : Mocking by Mockito

  - *RQ1.1 What are the most popular mocking frameworks?* This RQ helps identify mocking frameworks that practitioners could prioritize based on popularity.
  - *RQ1.2 How do the characteristics of projects impact the adoption of mocking frameworks?* We assume that project scale, activity, age, and domain may have an impact on the adoption of mocking frameworks. This provides empirical observation for practitioners regarding what type of projects are likely to use a mocking framework.

- ***RQ2****: What is the usage of mocking in Apache projects?* This RQ shed light on the intensity of mocking and what types of dependencies are usually mocked. It unfolds in two sub-RQs:

  - *RQ2.1: How intensively do the projects use mocking frameworks?* We assume that mocking is only practiced selectively by some developers since it comes with a cost. Here we aim to measure the intensity of mocking in the Apache projects to shed light on related future research. In addition, we also aim to measure how many developers in a project are usually engaged in using and maintaining mocking-related tests.
  - *RQ2.2: How often do developers mock external dependencies 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. This shed light on the motivation of using mocking.

- ***RQ3****: How has the usage of mocking in Apache projects evolved over time?* This RQ unfolds into three sub-RQs:

  - *RQ3.1: When did the projects first and last use mocking, and what has been the duration (in years) of the usage in the projects' history?* We aim to identify the first and the last years when a project was detected using mocking and calculate the years in between, which is the duration of usage. This provides an understanding of the overall history of mocking usage in the projects.
  - *RQ3.2: How has the usage intensity evolved over time?* Here, we measure and monitor the mocking usage intensity in each project in each year, and observe the trend of the usage intensity over the project history. This provides insights regarding whether and to what extent the mocking usage increases, decreases, stabilizes, or fluctuates with the growth and evolution of the projects.
  - *RQ3.3: How has the number of developers involved in mocking usage changed over time?* We estimate the number of developers who make contributions to mocking-related test files each year, and observe the trend over time. This helps us understand the dynamics of contributors to mocking usage and provides insights regarding the effort and resources dedicated to mocking in the projects.

- ***RQ4****: What are the most frequently used mocking APIs in Apache projects?* This RQ aims at revealing and analyzing the most frequently used APIs from the most popular mocking frameworks revealed in RQ1. The assumption is that only a portion of APIs are frequently used in practice. Revealing such could provide a learning cheat sheet for practitioners.
- ***RQ5****: Do developers always use a mocking framework when mocking is needed?* We observed that some test classes contain the keywords "mock" or "spy" in their names, implying that they relate to mocking, but do not use any mocking frameworks. The goal is to understand whether there are informal ways of mocking without relying on a mocking framework and to shed light on how to enhance mocking practice.

4 Study Process

This study focuses on the open source projects hosted on the Apache Software Foundation website (Apache software foundation projects list 2023), which contains a total of 246 Java projects for our study.We used the latest GitHub version as of October 2022. In order to answer the above research questions, our study process contains seven main conceptual steps illustrated in Fig. 3. Step 1: Basic Data Collection: We collect basic project information about the study subjects. This step prepares the raw data for the following-up steps. Step 2: Mock Frameworks Adoption Analysis (RQ1): We extract imported libraries in each project and manually verify whether a mocking framework is used. Step 3: Mock Usage Analysis (RQ2). We investigate to what extent mocking is practiced, and whether the mock objects are internal to a project, or external libraries or resources. Step 4: Mock Usage Evolution Analysis (RQ3), we select one snapshot each year from a project's history, and analyze the evolution history of mocking usage in the projects. Here we focus on the most popular frameworks identified in RQ1. Step 5: Mock API Analysis (RQ4). We extract and analyze the frequently used mocking APIs from the most popular mocking frameworks. Step 6: Sub-optimal Mock Identification (RQ5). We identify cases where developers leverage the concept of "mocking" without relying on a mocking framework. Step 7: Developers Survey which reflects on the RQs. We conduct a survey to gain a deeper understanding of the above research questions from the developers' perspective.

4.1 Step 1: Basic Data Collection

Firstly, we collect the source code of each project by cloning their git repository. In this study, we used the latest version of each project, released by October 2022. The details of which are listed here (List of releases for the project 2023). We find that 33 projects do not have a linked git repository. In addition, in 4 projects, we do not identify any test cases since these projects do not import JUnit at all. Next, we import each project to Eclipse and use *eclipse JDT* (Web 2013) to resolve the bindings among the software entities to prepare for the following-up

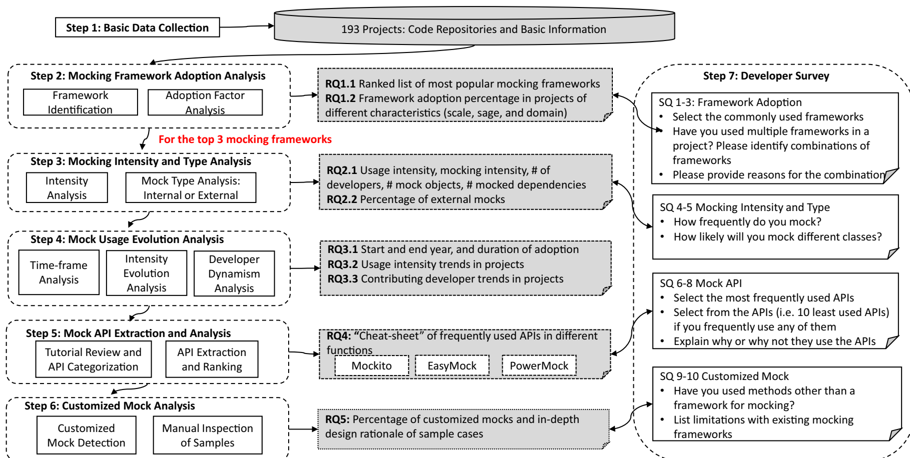


Fig. 3 : Study approach overview

analysis. This requires that the source code of a project be successfully compiled before being able to process by JDT, and thus we further excluded 16 projects that we cannot successfully configure and compile. 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 test LOC, the number of developers, and the number of available versions. Table 1 summarizes the average, median, maximal, and minimal of these measurements of the 193 projects.

Note that the number of test files is counted based on the import of *JUnit* library (JUnit 2023). If a *.java* file *imports* the *JUnit* APIs, we consider it as a test file for unit testing. The reason is that JUnit is the most commonly used framework for unit testing in Java language. Therefore, we assume that test files that import JUnit is for unit testing. As we will discuss in more detail in Section 8, this may lead to some inaccuracy in counting the number of unit test files, but should not impact the key findings and value of this study.

4.2 Step 2: Mocking Framework Adoption Analysis

In this step will first identify whether and which mocking framework(s) is used in each project; then analyze how different project characteristics may impact the adoption of mocking frameworks.

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 existing 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 namespace of each API. Next, we use a simple keyword, “mock”, to search for all potentially related framework names. To the best of our knowledge, the namespace 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 namespace. 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 namespaces 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.

Adoption Factor Analysis To help gain insights regarding how project characteristics contribute to the adoption of mocking frameworks, we first collect project measures, including the number of commits, the number of contributors, the number of files, LOC, the number of test files, test LOC, and the project duration. Then, we conducted a pair-wise correlation

Table 1 : Basic information of Mock usage empirical study subjects

Basic information	Avg.	Med.	Max.	Min.
#Java Files	1,537	738	20,760	17
#Test Files	524	189	11,064	1
LOC	165,443	76,517	1,574,022	993
Test LOC	61,579	17,414	784,965	0
#Developers	110	44	2,488	2
#Versions	56	36	584	0

analysis of these measures. The collected project measures were stored in an excel file, with the measures of each project in a row. The correlation analysis was done using Python Pandas library. We load the data using Python Pandas' read_csv() API. Lastly, we use the Pandas' corr() API to calculate the pair-wise correlation co-efficiency. The results are shown in Table 2. As we can see from the table that the top 6 of these 7 related factors have strong positive correlations with each other, with co-efficiency between 0.5 and 0.96. In comparison, project duration is not correlated with any of the other factors.

Therefore, in the rest of the study, we will only investigate how the project scale (measured by the number of Java files) and project duration impact the adoption of mocking frameworks, since the project scale strongly correlates with the other factors. We first divide the 127 into three ranks—as small projects (<400 files), medium projects (400-1500 files), and large projects (>1500 files). We make this division so that the number of projects in each rank is approximately the same. Next, we also divided the 127 projects based on their age—dated back to the year of the first code commit. As shown in Fig. 7c, the projects are in three groups: first commit before 2005, first commit between 2005-2010, and first commit after 2010. Note that the division is made so that the total number of projects in each bin is approximately the same. In RQ1.2, we will analyze how the adoption of mocking frameworks changes among projects of different scales and ages (i.e. duration).

Furthermore, we analyzed the tags associated with each project, provided by the official Apache website here Projects by category (2023), such as "big data", "sql", "search", etc. The tags capture the domain of each project based on its functions and content. There are a total of 30 unique tags provided on the website. The same tag may associate with different projects, and a project may also associate with multiple tags, such as "big data", and "cloud". To understand the impact of the project domain on the adoption of mocking frameworks clearly, we grouped these 30 tags into four high-level categories as described below.

- **Data Management** contains tags related to handling, managing, and securing various types of data (Briney 2015). For example, "big-data" deals with large data sets, "distributed-sql-database" relates to database management. There are 56 projects in this category.
- **Development & Infrastructure** contains tags related to the software development and infrastructure libraries (Myers et al. 2004). For example, "Build-management" involves compiling source code, "testing" covers software checks, and "library" relates to using existing libraries for development. There are 68 projects in this category.
- **Web Technologies** includes tags associated with online content management (Barker 2016). Tags like "html" and "xml" are used for structuring online content, and "content" and "web-framework" cover creating and managing digital content. There are 34 projects in this category. There are 34 projects in this category.
- **Networking & IoT** contain tags related to networking, cloud services, and the Internet of Things (Buyya and Dastjerdi 2016). For example, "Cloud" relates to online computing resources, "network-client" and "network-server" cover device communication over a network, and "iot" pertains to devices connected and exchanging data. There are 36 projects in this category.

4.3 Step 3: Mocking Intensity and Type Analysis

Intensity Analysis First, we collect three metrics that measure how intensely the projects use mocking and the developers contribute to the usage of mocking. For each project, we measure:

Table 2 : Correlation between different factors

	Commits	Contributors	.java Files	LOC	Test Files	Test LOC	Duration_days
Commits	1	0.74	0.81	0.76	0.75	0.71	0.02
Contributors	0.74	1	0.5	0.5	0.46	0.52	0.04
.java Files	0.81	0.5	1	0.92	0.96	0.9	-0.17
LOC	0.76	0.5	0.92	1	0.83	0.94	-0.17
Test Files	0.75	0.46	0.96	0.83	1	0.86	-0.14
Test LOC	0.71	0.52	0.9	0.94	0.86	1	-0.15
Duration_days	0.02	0.04	-0.17	-0.17	-0.14	-0.15	1

- UI, which stands for *Usage Intensity*, calculated as $|mockFiles|/|allFiles|$, where $mockFiles$ is the set of test files that use mocking, and $allFiles$ is the set of all the test files in the project. Thus UI measures what percentage of test files in a project use mocking.
- MI, which stands for *Mock Intensity* among test files that use mocking. It is calculated as $Avg(MockedDPs)/Avg(DPs)$, where $Avg(MockedDPs)$ is the average number of mocked dependencies of all the test files that use mocking, while $Avg(DPs)$ is the average number of all dependencies referenced by the test files. As such, MI measures, on average, what percentage of dependencies are mocked in test files with mocking.
- $\#d$, which counts the number of developers in each project who worked on test files with mocking.

Furthermore, we also investigate the intensity of mocking within individual test files that are identified with mocking. We calculate two metrics for each test file:

- $\#MO$, which counts the number of mock objects created in each test file.
- $\#MD$, which counts the number of dependencies being mocked in each test file.

Note that multiple mock objects may be created for the same dependency multiple times in a test file. Thus $\#MD$ is always $<= \#MO$ since the $\#MD$ only counts the number of mocked dependencies regardless of how many mock objects are created for each dependency.

Mock Type Analysis Next, we look into whether the mocked object is an external library or an internal function. 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 namespaces of the mocked dependencies. If a namespace is consistent with that of the project, it implies that the object is internal to the project. For example, in PDFBox (Apache PDFBox 2023), all the internal objects have this namespace, "org.apache.PDFBox". In comparison, an external dependency has a different namespace from the project. We assume that there may exist common external dependencies that developers need to mock, such as a database or an HTTP server. Therefore, we further count the most frequently mocked external dependencies.

4.4 Step 4: Mock Usage Evolution Analysis

This step aims to gain a deeper understanding of how mocking usage evolves over time in projects, focusing on the top three frameworks.

First of all, for each project, P, in our dataset, we retrieve the last release, P_i in each year i of its history. If there is no release in a particular year, we look for the first release of the next year, as such we make sure that two selected releases have roughly a year in between. We trace all the way back to the very first year of the project, as recorded by the git repository. Thus, we analyze a total of n snapshots of P over its entire available history on Git. The interval between two consecutive snapshots, P_{i-1} and P_i, is roughly 1 year based on our selection strategy.

Time-frame Analysis In RQ3.1, for each snapshot P_i of a project, we search for all the test files that import mocking framework APIs, by matching the framework namespaces. The matched file set is recorded as $mockFiles_{Pi}$, which contains all the test files with mocking. We record the first year, s, when the $mockFiles_{P_s}$ is detected not empty. It means that the project started using mocking in year s; similarly, we record the last year, e, when the

$mockFiles_{P_e}$ becomes empty after s. It means that the project stopped mocking in year e. Note that if e is never detected, it indicates that the project keeps using mocking in the time frame of our study, and thus $e = 2022$. We calculate the duration of mocking usage as $e - s$.

Intensity Evolution Analysis For RQ3.2, we calculate a mock usage intensity rate in each year of each project, as $Usage_Intensity_{p_i} = |mockFiles_{Pi}|/|allFiles_{Pi}|$, which describes what percentage of test files in P_i actually uses mocking. Next, we draw a trend line of how $Usage_Intensity_{p_i}$ changes over the years in each project. Furthermore, we manually review the trend lines and categorize the projects as increasing, decreasing, stable, or fluctuating usage intensity with the help of calculated regression models.

Developer Dynamism Analysis For RQ3.3, we record the number of developers, d_{p_i}, who worked on test files with mocking in each year P_i. Similar to RQ3.2, we draw trend lines of how d_{p_i} changes over the years, calculate the regression models and manually review the trend lines to categorize the projects into increasing, decreasing, stable, or fluctuating numbers of mocking-related developers.

4.5 Step 5: Mock API Extraction and Analysis

Here, we focus on the most popular mocking frameworks. It is of low value to investigate the APIs of an uncommonly used mocking framework. We first carefully review the official documentation and tutorials of the most popular mocking frameworks. This helps us to gain a systematic understanding of the API functions offered by different frameworks. Based on this understanding, we automatically extract all API usage instances from the Apache projects. The extraction is based on automatically searching for APIs calls that match the framework namespace. For example, Mockito's APIs all start with *org.mockito*. Lastly, we rank the APIs of the same function group from each mocking framework based on their frequencies being used across all the projects. We will present the most frequently used APIs in each function group. This empirical observation helps practitioners to prioritize APIs that are most frequently used in practice.

4.6 Step 6: Sub-optimal Mock Analysis

In this step, we aim to reveal whether developers leverage the concept of mocking without using any existing mocking frameworks. We call these classes Customized Mock Classes—meaning that developers used their own customized approach for potential mocking purposes. This RQ focuses on revealing what are the common types of Customized Mock Classes and whether they could be replaced by using a mocking framework, in particular Mokcito, since it is the most popular framework. It is possible that developers create mock objects using an informal approach, such as based on inheritance or by creating the mock objects manually (Wang et al. 2021; Wang 2021). We believe that these cases may point to the sub-optimal implementation of mock, due to various factors, such as a lack of related experience in using a mocking framework or even limitations with an existing framework.

The heuristic we leverage in this step is that, we identify test files with keyword "mock" or "spy" in their names, but does not import any APIs from existing mocking frameworks. We acknowledge that it is possible that developers may not always include the keyword "mock" or "spy" in such cases. Thus, we may not be able to retrieve all related cases using the searching heuristic. Once we identify such cases, we randomly select sample cases and review how developers use the concept of mocking without a mocking framework. For each Customized Mock Class, we examine their definition and how they support a test case. The high-level

purpose of a mocking framework is to help isolate test dependencies, i.e. replacing objects that the function under test depends on with "mock" objects. We first examine whether the Customized Mock Classes are intended to test dependency isolation, which aligns with the purpose of a mocking framework. For those that are not for dependency isolation, we inspect the class and summarize what is its intention. For those that are for dependency isolation, we analyze what is their design mechanism for achieving this purpose, and whether it is possible to replace them by using Mockito. In particular, Xiao et al. Wang et al. (2022) has discovered that inheritance is a common approach for mocking without using a mocking framework. Furthermore, Xiao et al. Wang et al. (2022) contributed an automated tool for replacing the inheritance mocking by using Mockito. We will apply Xiao's tool to our dataset. This helps us to understand gaps in open-source developers' expertise and even gaps in existing mocking frameworks.

4.7 Step 7: Developer Survey

This step conducts a survey involving developers who are likely to have experience with mocking. The objective is to gain a deeper understanding of the research questions by taking developers' input.

Participation Invitation Our target is developers who have prior experience with mocking. To effectively identify potential participants, we first download the entire Git revision logs of all the projects in our dataset. From the revision log of each project, we identify the names and emails of the developers who made commits to test files with mocking (according to our analysis of RQ2). We rank all the developers across all the projects based on the number of changes to mocking-related files. However, we noticed that the email contacts mined from the revision log are mostly affiliated with "apache.org", which are not the personal emails of the developers and are not likely to be the best way to reach them. Thus, we extend our invitation to the top 300 developers in order to possibly increase the number of responses we receive. Figure 4 shows that these 300 developers actually made almost 80% of commits

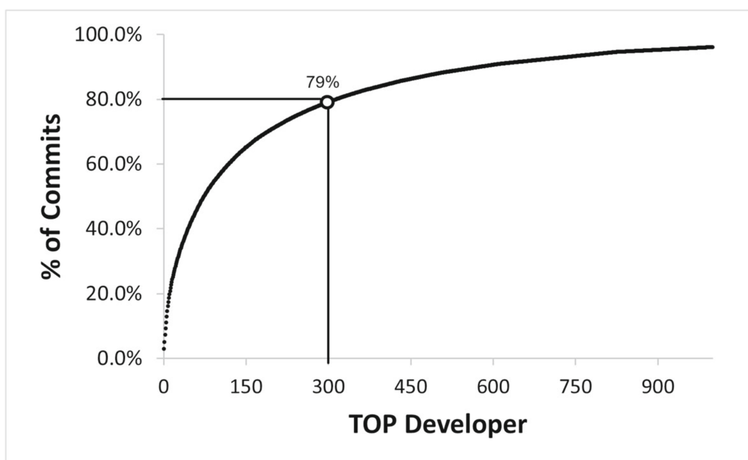


Fig. 4 : Cumulative distribution of top developers to Mocks

to the test files with mocks. Thus, this pool should quite comprehensively cover the most significant contributors to mocking. The survey is completely anonymous and does not collect participants' identities. As such, we have zero knowledge of who actually participated in the survey. The survey was initially sent on 5/25/2023 and results were collected on 7/15/2023, with three reminders in between.

Survey Questions We ask a total of 10 survey questions (SQs) grouped into four parts that correspond to the research questions so that we can get validation and additional insights from developers. We will elaborate on the survey questions and the rationale behind them:

- ***SQ 1-3: Framework Adoption:*** This part asks about developers' experience with commonly used frameworks and their combinations, and thus provides additional insights to RQ1:
  - *SQ1: Please select the commonly used frameworks.* Here we provide the list of mocking frameworks identified in our study and also provide an option for the participants to add new frameworks that are not listed.
  - *SQ2: Have you used multiple frameworks together in your project(s)? If yes, please check the ones that were used together.* Here we allow the participants to check multiple frameworks together.
  - *SQ3: If yes to SQ2, please talk about why multiple frameworks are used in your project(s)?* The goal is to obtain insights from developers, which could be subjective and cannot be comprehensively and accurately identified through purely quantitative repository mining in the RQ.
- ***SQ 4-5: Mocking Intensity and Type:*** This part investigates developers' perception of intensity and type of mocking, which maps to RQ2:
  - *SQ4: Based on your experience, please rate on how frequently do you mock in your project(s)?* Here we ask the participants to estimate the scale between 0% to 100% in two aspects in 1) the percentage of test files that use mocking; and 2) the percentage of dependencies being mocked. These are actually the Usage Intensity and the Mocking Intensity metrics (defined in Section 4.3) we used in RQ2. Also, we provide an option for the participants to add any additional information they would like to provide here.
  - *SQ5: How likely will you mock different classes?* Here we ask the participants to rate the likelihood of mocking internal and external classes, respectively, on a percentage scale (0% to 100%). We also provide an option for the participants to provide additional comments on this question.
- ***SQ 6-8: Most and Least Frequent Used APIs:*** This part examines developers' perception of the most and the least frequently used APIs of their primary framework.
  - *SQ6: Please select the most frequently used APIs.* Here we ask the developers to list the top 5 most frequently used APIs based on their experience.
  - *SQ7: Please select the ones from the APIs below that you frequently use.* We provide the least 10 frequently used APIs from each framework to the participants and ask them to check the ones that they actually frequently use. The objective is to confirm if their perceptions of the least frequently used APIs deviate from the mined results.
  - *SQ8: Can you explain why you did not use the ones from SQ7.* Here we ask the participants to share more detailed explanations for the ones that they did not choose in SQ7. This helps us understand why the APIs are less frequently used in practice.

– ***SQ 9-10: Experience with Customized Mock:*** This part aims to achieve additional insights regarding when and why developers use customized mocks presented in RQ5:
  – *SQ9: Have you used methods other than a framework for mocking? If so, can you describe the methods you have used?* We want to know if developers are aware of alternatives to mocking frameworks, especially if it is intentional to use customized mocks represented in RQ5.
  – *SQ10: Can you talk about limitations with existing mocking frameworks?* We aim to understand what limitations are encountered by developers to shed light on future research.

Of particular note, we did not design any questions that map to RQ3 (mocking evolution analysis). The reason is that evolution is long-term and high-level, of which individual developers may not have a direct perception. In specific, RQ3.1 focuses on objective facts of when a project starts to use mocking APIs, which was recorded by the repository. Individual developers join and leave the project and their knowledge represents their own experience, but not project history. RQ3.2 and RQ3.3 focus on the evolutionary trends, which represent the entire project, its history, and all its contributors as a whole. It would be biased to ask individual developers related questions.

5 Study Results

5.1 RQ1: Adoption of Mocking Frameworks

Overall, in the 193 projects, 127 (66%) projects use at least one mocking framework. This indicates that mocking frameworks are generally commonly used in Apache projects. Based on these 127 projects that use mocking frameworks, we further investigate two sub-RQs:

RQ1.1 What are the most popular mocking frameworks? Table 3 shows the popularity of different mocking frameworks. As we can see, among the 127 projects using mocking frameworks, Mockito is the most popular—used in 98 (77%) projects. EasyMock and PowerMock, ranking in second and third place, are used in 28% and 12% projects, respectively. As shown in the last column, the top 3 most frequently used mocking frameworks, namely Mockito, EasyMock, and PowerMock, together are used in 90% of projects.

It is worth noting that 47 (37%) projects use more than one mocking framework. Upon further investigation, we found that the number of adopted frameworks positively correlates

Table 3 : Popularity of Mocking frameworks

Mocking framework	#Projects	% Projects	Acc. % Projects
Mockito	98	74%	77%
EasyMock	36	28%	93%
PowerMock	16	12%	94%
jMock	11	8%	97%
Others	34	27%	100%
Total	127		
Multiple Mocking Frameworks	47 (37%)		

with the number of files, the LOC, and the number of commits, with co-efficiency of 0.71, 0.65, and 0.62 respectively. Thus, we imply that larger-scale and more complicated projects are more likely to adopt multiple mocking frameworks to leverage the complementary features. The majority of projects adopt two (in 33 projects) or three (in 9 projects) frameworks, and it is rare to adopt more than three frameworks.

The most common combination is Mockito and EasyMock in 10 projects. In 6 projects, EasyMock is the primary framework; and in 4 projects, Mockito is the primary framework. Primary framework means that the project uses this framework most of the time. Actually, Mockito and EasyMock offer comparable functions, except that Mocito supports spies, which is not available in EasyMock. We conjecture that Mockito and EasyMock are used together sinthe ce the different expertise/preferences of individual developers, as well as the migration between frameworks.

The second most common combination is Mockito and PowerMock, which appeared in 7 projects. This combination offers complementary functions from both frameworks. As discussed in RQ3, PowerMock typically offers more advanced functions to complement Mockito.

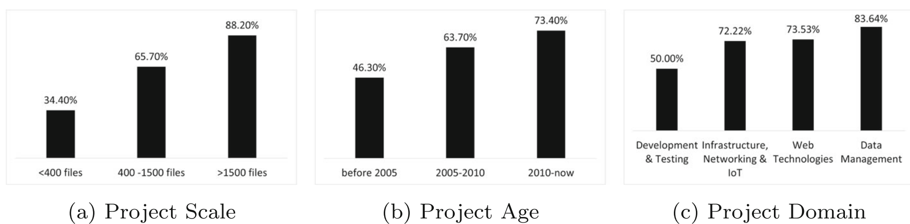
Less common combinations include EasyMock-PowerMock (3 projects) or Mockito-WireMock (3 projects). PowerMock provides advanced functions to complement EasyMock; while WireMock complements Mocito for mocking HTTP services. Thus, a project may use multiple frameworks as a result of different factors, including developers' preferences and expertise, migration of frameworks, as well as complementary functions.

RQ1.2 How do the characteristics of projects impact the adoption of mocking frameworks?

As we can see from Fig. 5a, 88% of large-scale projects, 65% of medium projects, and only 34% of small projects use mocking frameworks. The implication is that project scale is an important factor that contributes to the adoption of mocking frameworks. Large-scale projects see a stronger need for a mocking framework, compared to small-scale projects. This aligns with one's intuitive understanding since large-scale projects tend to have higher complexity in production dependencies that need the help of a mocking framework. As discussed above, we focus on the project scale, which strongly correlates with five other factors.

In addition, as shown in Fig. 7c, we observe that "younger" projects are more likely to adopt a mocking framework. 73% of projects "born" after 2010, 63% of projects "born" between 2005-2010, and only 46% of projects "born" before 2005 adopt a mocking framework. Note that the earliest release of Mockito dates back to 2008 (Mockito release notes 2023). Therefore, newer projects are more likely to adopt mocking frameworks.

Lastly, Fig. 5c shows the percentage of projects in different problem domains that adopted a mocking framework. Only 50% of "Development & Testing" projects use a mocking framework. In comparison, the majority of projects in the other three domain categories, i.e. 72% of "Networking&IoT" projects, 73% of "Web Technologies" projects, and 83% "Data Man-



(a) Project Scale (b) Project Age (c) Project Domain

Fig. 5 : % of framework adoption in projects of different characteristics

agement” projects, use mocking frameworks. This aligns with our findings to be presented in RQ2 that external dependencies, such as HTTP service and database, are more likely to be mocked, compared to internal dependencies in the projects.

RQ1: Take-aways:

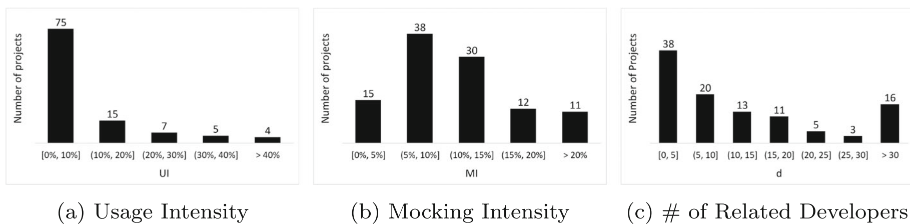
1. Mocking frameworks are widely us66%ed in Apache Java projects—66% of projects use a mocking framework.
2. The most popular mocking frameworks are Mockito, EasyMock, and PowerMock, which are used in 94% of projects collectively. Thus, it is recommended that practitioners should prioritize these three frameworks due to their popularity.
3. Larger-scale, more recent, and certain domain (e.g. networking, data) software projects tend to observe a stronger need in using mocking frameworks.

5.2 RQ2: Intensity and Type of Mocking

RQ2.1: How intensively do the projects use mocking frameworks? First, we show the overall distribution of mocking intensity in the projects in Fig. 6. This is measured in three aspects by *Usage Intensity (UI)*, *Mocking Intensity (MI)*, as well as the *# of Developers (#d)* (defined in Section 4.3), shown in Fig. 6a, b and c respectively.

From Fig. 6, we can observe that: 1) in most projects (70%), less than 10% of test files use mocking, implying that mocking is only practiced quite selected in some files. In comparison, in 9 projects, 30% to 60% of test files use mocking, implying more intensive usage. Upon further investigation, those with *UI* greater than 40% include *Tiles*, *CloudStack*, *Commons-DbUtils*, and *HttpComponents-Client*, targeting different areas such as web interfaces, databases, HTTP clients, and cloud fundamentals that are more likely to use mocking as shown in RQ1. 2) from the perspective of *MI*, in most projects (65%), between 5% to 15% of dependencies are mocked—also implying that mocking is overall quite selective. Lastly, 3) in most projects (53%), less than 10 developers worked on mocking, given that these projects could have hundreds of developers in total. The implication is that not all test files use mocking and not all dependencies are mocked. This points to future research direction for practitioners and researchers in investigating which dependencies should be mocked, and what characteristics of a dependency necessitate the usage of mocking.

Furthermore, Table 4 shows the distribution of test files with cummulative #MO (column 1 to column 3) and and #MD (column 4 to column 6). We find that the majority (58.5%) of mockFiles create three or fewer mocking objects. However, in 13.4% of test files with mocking, developers may create more than ten mock objects. In addition, when counting the number of mocked dependencies, in non-trivial test files (36.7%), only one dependency



(a) Usage Intensity (b) Mocking Intensity (c) # of Related Developers

Fig. 6 : Distribution of mocking intensity in projects

Table 4 : Cummulative distribution of #MO and #MD in test files

# Mock Objects	#Files (%)	Cumulative (%)	# Mocked Dps	#Files (%)	Cumulative (%)
1	1063 (27.7%)	27.7%	1	1328(36.7%)	36.7%
2	721 (18.8%)	46.5%	2	736(20.3%)	57.0%
3	460 (12.0%)	58.5%	3	461(12.7%)	69.7%
4	373 (9.7%)	68.2%	4	277(7.6%)	77.3%
5	222 (5.8%)	74%	5	212(5.9%)	83.2%
6	174 (4.5%)	78.5%	6	133(3.7%)	86.9%
7	117 (3.1%)	81.6%	7	115(3.2%)	90.0%
8	106 (2.8%)	84.4%	8	69(1.9%)	91.9%
9	87 (2.3%)	86.7%	9	58(1.6%)	93.5%
10+	512 (13.4%)	100%	10+	234(6.5%)	100.0%

Table 5 : External dependencies in Apache projects

(a) 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%

Table 5 : continued

(b) Most Frequently Mocked Library Classes.

Class name	Frequency
javax.servlet.http.HttpServletRequest	503
javax.servlet.http.HttpServletResponse	299
org.osgi.framework.Bundle	134
javax.servlet.ServletContext	114
java.util.Map	100
java.io.File	97
com.google.inject.Injector	96
org.apache.olingo.commons.api.edm.Edm	92
javax.persistence.EntityManager	85
com.google.inject.Provider	82

is mocked. In the remaining 63.3% (1-36.7%) test files, multiple dependencies are mocked. Particularly, in 6.5% of test files, more than 10 different dependencies are mocked in them. The implication is that there are often multiple dependencies that developers need to isolate, and multiple mock objects created, for unit testing, and thus multiple mock objects are created.

RQ2.2: How often do developers mock external dependencies in Apache projects? Table 5 shows the average, medium, maximal and minimal number (row 1) of mocked library dependencies in the Apache projects, as well as the percentage (row 2) among all mock objects. On average, the majority (61.1%) of the mocked objects are for isolating dependencies to external libraries. In some projects, this percentage reaches 100% (i.e. maximal). This indicates that mocking is an important way to isolate dependencies to external libraries.

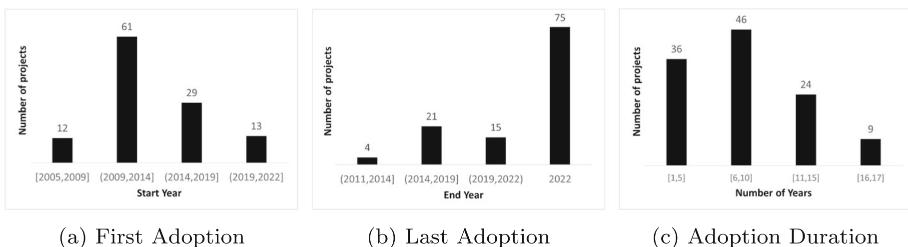
Furthermore, Table 5 lists the most frequently mocked external libraries, which are related to HTTP request/response, Database, and IO/File System. We imply that a key motivation for Apache developers to use a mocking framework is to 1) prevent interference from external libraries and 2) improve the testing performance by avoiding accessing a real HTTP server, database, or file system.

RQ2 Take-aways:

1. Mocking is practiced selectively. In most of the Apache projects, less than 10% of test files use mocking, less than 15% of dependencies are mocked, and less than 10 developers worked on mocking. It remains open research regarding when, where, and who should use mocking in testing practice.
2. 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 a key motivation for mocking is to isolate external functions from the function under test in Apache projects.

5.3 RQ3: Mocking Framework Usage Evolution

RQ3.1: When did the projects first and last use mocking, and what has been the duration (in years) of the usage in the projects' history? Fig. 7 shows the history of the adoption of the top three mocking frameworks in projects. Figure 7a shows the distribution of when the projects first started to adopt a mocking framework; Fig. 7b shows the distribution of when the last code commit that relates to the mocking framework was made; Fig. 7c shows the distribution of the duration, measured by the time between the first adoption and the last related commit, of mocking framework adoption in the projects.



(a) First Adoption (b) Last Adoption (c) Adoption Duration

Fig. 7 : Adoption history of the top 3 mocking frameworks

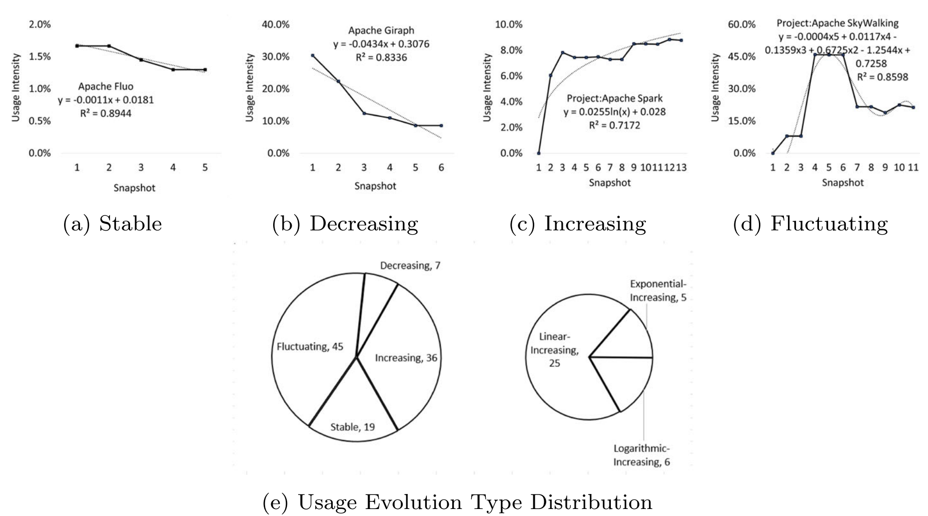
From Fig. 7a, we can observe that 12 projects are early adopters, which started using a mocking framework between 2005 and 2009, almost two decades ago. The majority, 61, projects started using a mocking framework between 2009 and 2014, about a decade ago from the time of this writing. There are also “laggers”, i.e. 29 projects first started to use mocking frameworks between 2014 and 2019 (less than a decade); while the remaining 13 projects first started to use mocking frameworks between 2019 and 2022 in recent 5 years.

As shown in Fig. 7b, the majority of the projects, 75 out of 115, are still actively using mocking frameworks in 2022, when is our study dated back to. In comparison, the other projects stopped using a mocking framework in an earlier year. For example, in 15 projects, no usage of mocking frameworks is detected in these projects between 2019 and 2022.

Finally, Fig. 7c shows that the largest group with 46 projects have been using a mocking framework for 6 to 10 years; 36 projects have been using a mocking framework for 5 years or less; 33 projects have been using a mocking framework for more than 10 years.

RQ3.2: How has the mocking usage intensity evolved over time? Figure 8 shows examples (Fig. 8a, b, c and d) and the overall distribution of different types of mocking framework usage evolution patterns (Fig. 8e) we observed among the projects which have been detected to use a mocking framework. We observed that, during the traceable evolution history of a project, the mocking framework adoption intensity (i.e. percentage of test files that involve using mocking framework APIs) may remain stable, decrease, increase, or fluctuates over the years of the history.

Figure 8a shows the trend of the adoption intensity in Apache Fluo remains quite stable over five snapshots (i.e. each snapshot represents one year in its traceable history on Git). According to Fig. 8e, 19 out of the 109 projects (note that here we only consider projects that used the top 3 mocking frameworks) that have used a mocking framework show this stable usage. There could be two possible scenarios behind these projects: 1) the project itself has been stable over the years and so does the mocking framework usage; 2) the usage of mocking frameworks grows consistently with the growth of the project.



(a) Stable (b) Decreasing (c) Increasing (d) Fluctuating

(e) Usage Evolution Type Distribution

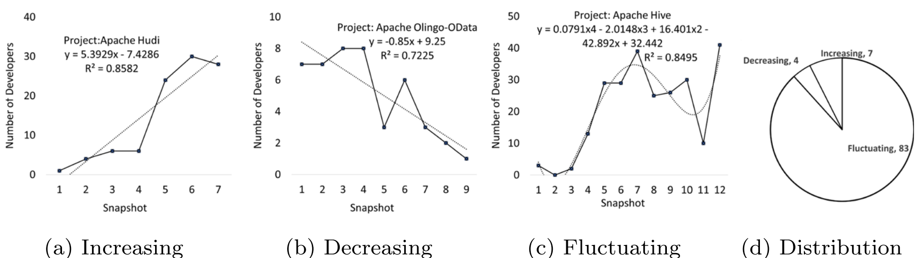
Fig. 8 : Mocking usage intensity over time: examples and distribution

Figure 8b shows the trend in Apache Giraph, where about 30% of the test files in it involve mocking framework APIs in a snapshot; while as the project grows over the years, only about 10% of the test files use mocking framework APIs in year 6. As a reminder, as introduced in RQ 2.1, 10% is the average adoption intensity over the projects. We conjecture that a decreasing trend is a possible indication of a potential lack of support in mocking usage to match the pace of growth of the project. This could be caused by various factors, such as lack of mocking experts, time pressure, higher priorities in the projects, etc.

Figure 8c shows the opposite trend in Apache Spark that the mocking usage intensity increases over time—from 0% in year 1 to 9% in year 13, with fast increases between year 1 and year 3, and slow-down and stabilize afterward. This example represents a typical scenario that a project quickly picks up using mocking framework over the years and maintains its usage stably for its needs. As shown in Fig. 8e, a non-trivial number of projects, 36 (out of 109) projects, show an increasing usage over time. In particular, the increasing trends are further observed in three types as shown on the right side pie-chart of Fig. 8e. They are: linear increasing in 25 projects —indicating continuous and ongoing increasing usage; exponential increasing in 5 projects—representing ever-faster increasing usage; or logarithmic in 6 projects—fast increasing first and stabilizing later, such as shown in Fig. 8c. Overall, an increasing trend indicates a growing commitment in using and maintaining mocking framework usage that matches the projects' interests. While, the different slopes of trends (i.e. linear, exponential, and logarithmic) may entail more detailed factors behind projects that incentivize the usage of mocking frameworks to different levels in different phases. We will leave a more in-depth analysis of these factors in our future research.

Lastly, Fig. 8d shows that, in Apache SkyWalking, there were some fluctuations in the usage intensity of mocking frameworks. Between year 3 to year 4, the usage intensity increases drastically from 7% to 45%; while between year 6 to year 7, it drops from 45% to 18%. Upon further investigation of SkyWalking's revision history, especially in year 3 and year 7, we found that the fluctuation is associated with the focus on the project. In year 3, the project added a significant number of mock files because of updates and fixes to the local/exit span operation name register mechanism and client-side endpoint register in the service mesh, so the overall usage intensity increases this year. Year 7's release records showed that SkyWalking implemented a number of fixes and upgrades in areas such as storage and query optimization and the Kubernetes Java client. Based on our conclusions in RQ2, we found that databases, filesystems, and HTTP web servers were more likely to use mock. Therefore, we believe that these function changes resulted in a large number of deletions of mock-related files. This led to a decrease in the usage intensity.

RQ3.3: How has the number of developers involved in mocking usage changed over time? Figure 9 shows how the numbers of mocking-related contributors change over time in the



(a) Increasing (b) Decreasing (c) Fluctuating (d) Distribution

Fig. 9 : Involved developers over time: examples and distribution

history of the projects. Similar to above, here we show examples of different types of trends of: 1) increasing number of contributors in Fig. 9a, 2) decreasing number of contributors in Fig. 9b, and 3) fluctuating number of contributors in Fig. 9c. In addition, we also show the overall distribution of the different types of trends in Fig. 9d.

From Fig. 9d, we can see that the significant majority, 83 out of 94, projects have a fluctuating number of contributors who work on mocking-related files over time. Note that the total number of projects is 94 here since we only consider projects that 1) used the top 3 mocking frameworks in their history and 2) had at least 3 developers. The implication is that the number of developers who actually worked on or maintained the usage of mocking frameworks is quite dynamic in the projects, possibly depending on various factors, such as the stability of the test cases that rely on the mocking functions, as well as developers' availability and project phase and priority. In 4 projects, the number of involved developers decreases over time; while in the other 7, the number increases over time. For example, Fig. 9a shows the number of related developers in Apache Hudi, which steadily increased from 0 in year 1 to about 30 developers in year 7—indicating robustly increasing commitment of the development team on mocking. In comparison, Fig. 9b shows that, in Apache Olingo-Odata, the number of developers who work on mocking functions decreases from 7 in year 1 to 1 in year 9, indicating reduced effort in using/maintaining mocking functions.

RQ3 Take-aways:

1. Most (53%) projects started to use mocking frameworks between 2009 and 2014; early adopters (10%) started between 2005 and 2009; and "laggers" started within the past decade (25% projects) or five years (11% projects). The majority of projects (65%) are still using mocking frameworks as of 2022 when this study was conducted. Most projects (69%) accumulated 6 or more years and 29% of projects accumulated more than 10 years of experience, in using mocking frameworks.
2. The usage intensity of mocking functions in different projects shows different trends in the projects' evolution history, i.e. increasing, decreasing, stable, and fluctuating—implying the compound effects of various factors, such as the pace of a project's growth, the available resources, time pressure, and project priority. More specifically, 42% of projects experienced fluctuating usage over time—implying that different factors may come to play at different phases of the projects. 33% of projects experienced increasing usage—implying an increasing need for mocking. 18% of the projects showed stable usage over time—implying balanced use of mocking with the growth and evolution of the project. Finally, a small portion (7%) of projects experienced decreasing usage of mocking frameworks—indicating a possible insufficient mocking that falls behind the pace of project growth.
3. In most projects (88%), the number of developers who worked on or maintained mocking-related test files is quite dynamic over the projects' history. This indicates such the effort required and/or the available resource for supporting mocking could be quite dynamic in Apache projects.

5.4 RQ4: Most Frequently Used Mocking APIs

This RQ focuses on analyzing the most frequently used APIs from the three frameworks, Mockito, EasyMock, and PowerMock. We found that there are a total of 317 APIs in Mockito, 278 APIs in EasyMock, and 311 APIs in PowerMock, which are the three most popular

mocking frameworks. But only 109 (34.5%), 68 (24.5%), and 75 (24.1%) APIs in these three frameworks, respectively, are actually used by Apache projects. The implication is that developers who want to maximize the learning outcomes of how to use a mocking framework should start from the ones that are more frequently used, and leave the ones that are not likely to be used in practice.

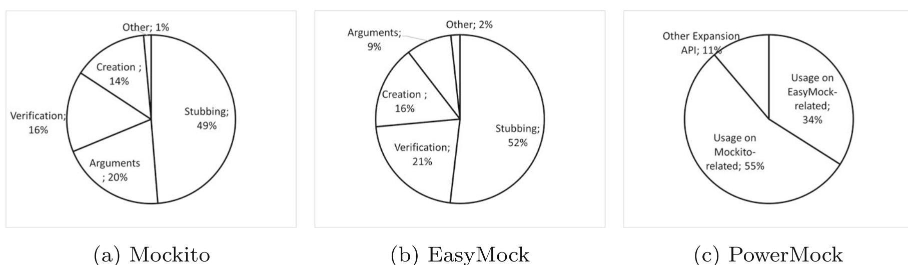
RQ4.1: What are the most frequently used APIs in Mockito? After reviewing the official tutorial of Mockito, we found that its APIs are generally in five categories for:

1. Creating mock objects (3/317, 1% APIs);
2. Stubbing the behavior of the mock objects(36/317, 11% APIs);
3. Verifying the execution of mock objects (40/317, 13% APIs);
4. Managing the arguments of mock objects before or after its execution (54/317, 17% APIs);
5. Other more specific behaviors. For example, initMocks(), isSpy() and isMock() are getting and controlling the mock objects, which do not belong to any of the first four types (184/317, 58% APIs);

Figure 10a shows the distribution of the usage of APIs in these five categories based on Apache projects. As we can see that, stubbing APIs have the highest usage— covering 49% API calls; creation, verification, and argument APIs are used roughly equally, in 14%, 16%, and 20% of the total API calls. Other APIs have used rarely only 1% API calls. The implication is that developers extensively leverage the stubbing APIs to set up the behavior of the mocked object.

Table 6 list the most frequently used APIs for verification, creation, method stubbing, managing arguments. As we can see that the top five APIs in each group usually cover the majority of usage. Thus, developers can refer to these APIs as a quick cheat-sheet for learning how to use Mockito APIs.

RQ4.2: What are the most frequently used APIs in EasyMock? Based on the official tutorial of EasyMock, its APIs are in the same types as Mockito as: 1) Creating mock objects; 2) Stubbing the behavior of the mock objects; 3) Verifying the execution of mock objects; 4) Managing the arguments of mock objects before or after its execution; and 5) Other behaviors. Figure 10b shows the distribution of the usage of the five types of APIs, which shows highly consistent distribution as Mockito (see Fig. 10a). The Stubbing APIs are used most frequently in 50% cases. Considering the observations in Fig. 10b and a, we can imply that manipulating the behavior of mock objects is the most significant part of using mocking frameworks. This suggests that developers frequently manipulated different behaviors for the same mock object.



(a) Mockito (b) EasyMock (c) PowerMock

Fig. 10 : API type usage distribution in Mockito, EasyMock, and PowerMock

Table 6 : Top five APIs in each category in Mockito

Top five stubing APIs In Mockito			
API	Frequency	Percentage	Description
thenReturn	15,513	74.96%	Stub the return value for a non-void method
doReturn	2,191	85.55%	Stub the return value for any method.
doNothing	389	87.43%	Setting void methods to do nothing.
doAnswer	354	89.14%	Stub a method with generic Answer
thenThrow	323	90.70%	Sets a Throwable type to be thrown when the method is called
others	1925	100.00%	API for all other stubber behaviors
Top five verification APIs In Mockito			
API	Frequency	Percentage	Description
verify	7,726	61.08%	Verify a specific behavior of the mock object
times	3,721	90.50%	Verify exact number of invocations for the methods of a mock object
never	660	95.72%	Making sure interaction(s) never happened on mock
verifyNoMore-Interactions	174	97.09%	Checks if any of given mocks has any unverified interaction.
atLeastOnce	104	97.91%	Verifies that there is at least 1 invocation during the given period.
other	264	100.00%	API for all other verification behaviors

Table 6 : continued

Top five stubing APIs In Mockito			
API	Frequency	Percentage	Description
Top five arguments APIs In Mockito			
API	Frequency	Percentage	Description
any	6,332	38.75%	Matches anything, including nulls and varargs.
eq	2,592	54.61%	Argument that is equal to the given value.
anyLong	1,816	65.72%	Any long or non-null Long.
anyString	1,439	74.53%	Any non-null String
capture	587	78.12%	Use it to capture the argument.
other	3575	100.00%	API for all other Arguments behaviors
Top five creating Mocks APIs In Mockito			
API	Frequency	Percentage	Description
mock	10,795	92.54%	Create an empty mock object.
spy	843	99.77%	Create a spy object
mockStatic	27	100.00%	Creates a thread-local mock controller for all static methods

We can see in Table 7 the top five most frequently used APIs of each type of APIs in EasyMock. We can make consistent observations that the top few APIs usually are used extensively, and thus practitioners could focus on these few APIs to accelerate the learning process.

RQ4.3: What are the most frequently used APIs in PowerMock? PowerMock is an extension to EasyMock and Mockito PowerMock framework site (2023). Functionally, it can be divided into Mockito extensions, EasyMock extensions, and other general APIs that can be used to support both Mockito and EasyMock. Figure 10c shows the usage distribution of these three types of APIs: 55% usage on Mockito-related APIs, 34% usage on EasyMock-related APIs, and 11% usage on other general APIs. Of particular note, as shown in Table 3, EasyMock is only used in 28% projects, while Mockito is used in 74% projects. This aligns with the fact that EasyMock-supporting APIs have lower usage than Mockito-supporting APIs.

Table 8 summarizes the top five most frequently used APIs in each of the three groups from PowerMock. Of particular note, PowerMock contains APIs that match the original APIs from Mockito or EasyMock for which it provides support. For example, the *when* in Mockito is replicated in PowerMock's namespace to be used together with other extension APIs. Here, we only list the top five extension APIs since the replicated APIs have exactly the same functions as in the original framework. We can reach the same observation as in RQ-2.1 (Mockito) and RQ-2.2 (EasyMock) that the top five APIs in each category usually merit the majority of usage in PowerMock. Thus, one can refer to this table as a quick cheat sheet for getting started on PowerMock. For an additional note, we also observed that a key extended usage of PowerMock relates to supporting mocking static objects (such as *mockStatic* and *verifyStatic*), which is not supported in the original Mockito or EasyMock.

RQ4 Take-aways:

1. Among the total 317 APIs in Mockito, 278 APIs in EasyMock, and 311 APIs in PowerMock, which are the three most popular mocking frameworks, only 34.5%, 24.5%, and 24.1% APIs, respectively, are actually used by Apache projects.
2. The Mockito and EasyMock APIs mainly provide four types of functions for creating mock objects, stubbing behaviors, verifying execution, and managing arguments of mock objects. While PowerMock provides extensions to Mockito and EasyMock APIs.
3. The top five APIs in each functional type of the three mocking frameworks usually take the majority (78% to 100%) of usage in Apache projects. This indicates that developers can focus on these APIs to quickly learn the common usage of these mocking frameworks. Tables 6, 7, and 8 provides a quick cheat-sheet to these APIs for practitioners.

5.5 RQ5: Mocks without Leveraging Mocking Frameworks

We identified all the Customized Mock Classes in the 193 Apache projects. We found that 143 out of the 193 projects include Customized Mock Classes. Table 9 shows the comparison of the number of classes using *Customized Mocks* vs. that using mocking frameworks. The first row shows that there are a total of 2,237 Customized Mock Classes in these 143 projects, as well as the average, maximal, minimal, and median, of the number of *Customized Mock Classes*, identified across the 143 projects. In addition, the second row shows the number

Table 7 : Top Five APIs in each category in EasyMock

Top Five Stubbing APIs in EasyMock			
API	Frequency	Percentage	Description
andReturn	12,134	94%	Sets a return value that will be returned for the expected invocation
expectLastCall	886	97%	Returns the expectation setter for the last expected invocation in the current thread
andStubReturn	341	98%	Sets a stub return value that will be returned for the expected invocation
reset	165	99%	Will reset capture to a “nothing captured yet” state
andThrow	154	99%	Sets a throwable that will be thrown for the expected invocation
other	153	100%	API for all other API for Stubbing behaviors
Top Five Verification APIs in EasyMock			
API	Frequency	Percentage	Description
anyTimes	5,951	53.84%	Expect the last invocation any times
verify	1,461	67.06%	Verifies that all expectations were met and that no unexpected call was performed
atLeastOnce	1,434	80.03%	Expect the last invocation at least once
once	1,419	92.87%	Expect the last invocation once.
verifyAll	413	96.61%	Verifies all registered mock objects have their expectations met and that no unexpected call was performed
times	375	100.00%	Expect the last invocation any times
Top Five Arguments APIs in EasyMock			
API	Frequency	Percentage	Description
anyObject	1,458	33%	Expects any Object argument
eq	1,069	57%	Expects that is equal to the given value
capture	508	68%	Expect any object but captures it for later use
newCapture	443	78%	Create a new capture instance that will

Table 7 : continued

Top Five Stubbing APIs in EasyMock			
API	Frequency	Percentage	Description
			keep only the last captured value.
anyString	285	85%	Expect any string whatever its content is.
other	676	100%	API for all other Arguments behaviors
Top Five Creating Mocks APIs in EasyMock			
API	Frequency	Percentage	Description
createNiceMock	3,871	47%	Create an empty mock object with default returns
createMock	3,571	91%	Create an empty mock object without default returns
createStrictMock	464	97%	Creates a mock object that implements the given interface, order checking is enabled by default
createMockBuilder	136	99%	Create a mock builder allowing to create a partial mock for the given class or interface
niceMock	55	99%	Creates a mock object that implements the given interface, order checking is disabled by default, and the mock object will return 0, null or false for unexpected invocations
other	57	100%	API for all other API for Creatmock behaviors

Table 8 : Distribution of PowerMock's most popular APIs

Top Five Mockito-related APIs in PowerMock			
API	Frequency	Percentage	Description
mockStatic	180	44%	Enable static mocking for all methods of a class.
whenNew	63	59%	Stub behavior for constructor.
verifyStatic	61	74%	Verifies certain behavior of the mockedClass
withArguments	38	83%	Specify input arguments for constructor stubbing.
withAnyArguments	38	93%	Accept any input arguments for constructor stubbing.
other	30	100%	All other APIs
Top Five EasyMock-related APIs in PowerMock			
API	Frequency	Percentage	Description
mockStatic	52	66%	Enable static mocking for a class
expectNew	14	84%	Stub behavior for constructor.
expectPrivate	5	90%	Stub behavior for private static methods
mockStaticPartial	2	92%	Mock a single static method
createStrictMock-AndExpectNew	2	95%	Convenience method for createStrictMock followed by expectNew
other	4	100%	All other APIs
Top Five Ohter Support APIs in PowerMock			
API	Frequency	Percentage	Description
setInternalState	75	42.13%	Set the value of a field using reflection.
suppress	25	56.18%	Suppress a specific method
methods	14	64.04%	Get an array of Method's that matches the supplied list of method names
getInternalState	12	70.79%	Get the value of a field using reflection
method	11	76.97%	Get a method without having to specify the method name.
others	41	100.00%	Other methods of expansion

Table 9 : # classes using customized mocks vs. using mocking frameworks

	Avg.	Max.	Min.	Mid.	Total
# Customized Mock Classes	16	400	1	6	2237
Classes Using Customized Mocks	81	6066	1	6	9697
Classes Using Mocking Frameworks	201	4828	1	15	28792
% of Customized Mock Usage	44%	100%	0%	27%	25%

of test classes that use the *Customized Mocks*. For example, a total of 9,697 test classes use the 2,237 *Customized Mocks*. This indicates that a *Customized Mock Object* could be used by multiple (average 4 = 9,697/2,237) test classes. In comparison, the third row shows the number of classes that use mocking framework APIs. That is, a total of 28,792 test classes across the 143 projects use mocking framework functions for mocking. Thus, the last row highlights the percentage of *Customized Mock* usage across projects (i.e. classes that use either *Customized Mock* or framework together as the base). On average, 44% of classes in a project may use *Customized Mocks*. While across the 143 projects, a total of 25% test classes use *Customized Mocks*. Thus, we conclude that Customized Mock Classes are not rare, and they are non-trivially used by test classes in practice. This motivates us to investigate the internal design of the Customized Mock Classes.

To gain an in-depth understanding of how the Customized Mock Classes, we randomly sampled 2% classes from the total 2,237 Customized Mock Classes—this provides us a sample dataset of 44 such classes. Actually, we started the analysis by randomly sampling about 1% classes and then increased the sample to 2%, which did not reveal any new scenarios and types of Customized Mock Classes. Thus, we stopped at 2%. Though we acknowledge that we cannot guarantee that this has covered all possible scenarios of customized mock classes, our analysis should have revealed the main scenarios that developers should be aware of. Following the rationale described in Section 4, we classify the 44 sample cases as shown in Fig. 11.

For 40 Customized Mock Classes (the left child of the root node in Fig. 11), we confirmed that these classes are indeed created for test dependency isolation. Namely, we were able to identify which dependency they are trying to isolate in test cases. Interestingly, all of

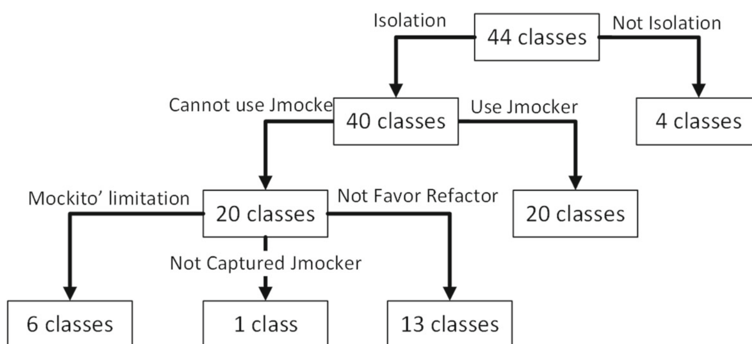


Fig. 11 : Classification of the 44 sample classes

```java
1  // Test target class
2  public class ProductService {
3      public List<Product> sortProducts(List<Product> products) {
4          Collections.sort(products,
5          Comparator.comparingInt(Object::hashCode));
6          return products;
7      }
8  }
9  // Test dependent production class
10 public abstract class Product {}
11 // Mocked dependency
12 class MockProduct extends Product {
13     int hashCode;
14     public MockProduct(int hashCode) {
15         this.hashCode = hashCode;
16     }
17     @Override
18     public int hashCode() {
19         return this.hashCode;
20     }
21 }
22 // Test class
23 public class TestProductService {
24     @Test
25     public void testSortProducts() {
26         Product product0 = new MockProduct(0);
27         Product product1 = new MockProduct(1);
28         ProductService productService = new ProductService();
29         List<Product> products = Arrays.asList(product1, product0);
30         productService.sortProducts(products);
31         assertEquals(product0, products.get(0));
32         assertEquals(product1, products.get(1));
33     }
34 }
```

(a) Override JDK APIs

```java
1  // Test target class
2  public class ProductService {
3      public List<Product> copyProducts(List<Product> products) {
4          final List<Product> copied = new ArrayList<>();
5          products.stream().forEach(p -> copied.add(p.deepCopy()));
6          return copied;
7      }
8  }
9  // Test dependent production class
10 public abstract class Product {
11     abstract Product deepCopy();
12 }
13 // Mocked dependency
14 class MockProduct extends Product {
15     public int hashCode;
16     public MockProduct(int hashCode) {
17         this.hashCode = hashCode;
18     }
19     @Override
20     public Product deepCopy() {
21         return new MockProduct(hashCode);
22     }
23 }
24 // Test class
25 public class TestProductService {
26     @Test
27     public void testCopyProducts() {
28         Product product0 = new MockProduct(0);
29         Product product1 = new MockProduct(1);
30         ProductService productService = new ProductService();
31         List<Product> products = Arrays.asList(product0, product1);
32         productService.copyProducts(products);
33         assertEquals(product0.hashcode, products.get(0).hashCode);
34         assertEquals(product1.hashcode, products.get(1).hashCode);
35         assertNotSame(product0, products.get(0));
36         assertNotSame(product1, products.get(1));
37     }
38 }
```

(b) Self-Reference

Fig. 12 : Customized mock examples: override JDK and self-reference

them involve using inheritance for test dependency isolation. That is, the developer creates a sub-class of a production class and uses method overriding to control the behavior for testing purposes. In the test case, whenever the parent class should be used, the sub-class is actually used so that the test case no longer depends on the parent class (i.e. test dependency isolation). We applied Xiao et. al's tool on the 40 classes. We found that the tool is applicable to 20 (50%) cases. The remaining 20 cases are not able to apply Xiao et. al's tool because of three general reasons: 1) they are related to the limitations of Mockito; 2) the refactoring is not necessary or beneficial; and 3) the tool has its limitation to handle the particular case. Following, we list the details regarding the 20 cases:

– Six classes cannot be refactored because of the limitation of Mockito:

```java
1  // Test target class
2  public class ProductService {
3      public List<Product> filterProducts(List<Product> products, Tag productTag) {
4          return products.stream()
5          .filter(p -> p.getProductTags().contains(productTag))
6          .collect(Collectors.toList());
7      }
8  }
9  // Test dependent production class
10 public abstract class Product {
11     protected abstract Set<Tag> getProductTags();
12 }
13 public enum Tag {
14     FOOD, BEVERAGE
15 }
16 // Mocked dependency
17 class MockProduct extends Product {
18     public Tag tag;
19     public MockProduct(Tag tag) {
20         this.tag = tag;
21     }
22     @Override
23     protected Set<Tag> getProductTags() {
24         return Set.of(tag);
25     }
26 }
27 // Test class
28 public class TestProductService {
29     @Test
30     public void testFilterProducts() {
31         Product product0 = new MockProduct(Tag.FOOD);
32         Product product1 = new MockProduct(Tag.BEVERAGE);
33         ProductService productService = new ProductService();
34         List<Product> products = Arrays.asList(product0, product1);
35         List<Product> filterProducts = productService.filterProducts(products, Tag.FOOD);
36         assertEquals(1, filterProducts.size());
37         assertEquals(product0, products.get(0));
38     }
39 }
```

Fig. 13 : Customized mock example: override protected method

```java
1  // Test target class
2  public class ProductService {
3      public double getProductPrice(Product product, double deliveryDistance) {
4          if (product instanceof Carriage) {
5              return product.getPrice() +
6              ((Carriage) product).getUnitDeliveryPrice() * deliveryDistance;
7          }
8          return (double) product.getPrice();
9      }
10 }
11 // Test dependent production class
12 public interface Carriage {
13     double getUnitDeliveryPrice();
14 }
15 public abstract class Product {
16     abstract int getPrice();
17 }
18 // Mocked dependency
19 class MockProduct extends Product implements Carriage {
20     @Override
21     public int getPrice() {return 10;}
22     @Override
23     public double getUnitDeliveryPrice() {return 1.0;};
24 }
25 // Test class
26 public class TestProductService {
27     @Test
28     public void testGetProductPrice() {
29         Product product = new MockProduct();
30         ProductService productService = new ProductService();
31         double price = productService.getProductPrice(product, 20.0);
32         assertEquals(30, price);
33     }
34 }
```

(a) Inherit Multiple Classes

```java
1  // Test target class
2  public class ProductService {
3      public Product decorateProduct(Product product) {
4          return product.decorate();
5      }
6  }
7  // Test dependent production class
8  public abstract class Product {
9      abstract Product decorate();
10     abstract Product addPrice(int price);
11 }
12 // Mocked dependency
13 class MockProduct extends Product {
14     ProductDecoratorSupplier productDecoratorSupplier;
15     int price;
16     public MockProduct(ProductDecoratorSupplier productDecoratorSupplier, int price) {
17         this.productDecoratorSupplier = productDecoratorSupplier; this.price = price;
18     }
19     @Override
20     Product decorate() {return productDecoratorSupplier.decorate(this);}
21     @Override
22     Product addPrice(int price) {
23         this.price += price;
24         return this;
25     }
26     interface ProductDecoratorSupplier {
27         Product decorate(MockProduct product);
28     }
29 }
30 // Test class
31 public class TestProductService {
32     @Test
33     public void testDecorateProduct() {
34         Product product = new MockProduct((p) -> p.addPrice(20), 10);
35         ProductService productService = new ProductService();
36         Product newProduct = productService.decorateProduct(productService.decorateProduct(product));
37         assertEquals(50, newProduct.price);
38     }
39 }
```

(b) Inner Class

Fig. 14 : Customized mock examples: inherit multiple classes and inner class

- Two classes override JDK APIs, and thus cannot be replaced by Mockito, since Mocktio's implementation depends on JDK APIs. For instance, in Fig. 12a, test target *ProductionService* has a method to sort products based on the product's *hashCode* (lines 1-8). In order to test *sortProducts* method, the developer creates a *MockProduct* which extends *Product* and *override hashCode*, which is a JDK API (line 11-21). In the test case *testProductService*, the developer instantiates 2 *MockProducts* (line 26-27), call *sortProducts* (line 30), and assert the order of the product list on line 31-32. *Mockito* is built upon the two JDK APIs, namely *equals()* and *hashCode()* [1]. Mocking the behavior of these two APIs will endanger the normal functions of Mockito.
- Two classes have self-reference, but Mockito cannot handle self-reference. For example, in Fig. 12b, test target *ProductionService* has a method *copyProducts* to deep copy a list of products (line 2-8). In order to test this method, the developer creates a MockProduct to extend the *Product* and override *deepCopy* to create an instance of itself, leading to the self-referencing of *MockProduct* (lines 20-22). In the test case, the developer creates a list containing 2 *MockProducts* (line 31) and calls the test target *copyProducts* (line 32). Then the developer further asserts the copied products are identical to the original products (lines 33-34) and verifies that they are not the same instance (lines 35-36).
- Two classes override a protected attribute/method. Mockito does not support accessing protected attributes/methods. But one could mitigate this by using PowerMock. For instance, as shown in Fig. 13, *ProductService* has a method *filterProducts* to filter the given list of products based on the given product tag (line 1-8). In order to test this method, the developer creates a subclass of *Product*, named *MockProduct*, to override *getProductTags*, which is *protected* (line 16-26). Then in the test case, the developer creates a list of mocked products as a parameter for *filterProducts* (line 35). The developer further verifies the filtered list size and product instance (lines 36-37).

[1] https://github.com/mockito/mockito/wiki/FAQ#what-are-the-limitations-of-mockito

- In 13 classes, the refactoring is not beneficial or necessary:
  - Two classes are not instantiated anywhere and thus refactoring is not necessary. Due to its simplicity, we do not provide a code snippet example here.
  - Six classes inherit/implement multiple production classes (PowerMock framework site 2023), indicating they mock the behavior of multiple objects. It is a general design principle not to mock multiple different classes when using a mocking framework. For instance, as illustrated in Fig. 14a, *ProductService* has a method *getProductPrice* (line 1-10) to calculate a product's label price based on the import price and delivery cost. If a product is a *Carriage*, meaning it was shipped from another location, the label price needs to include the shipping cost. To test this function, the developer creates a *MockProduct* to extend *Product*, which then implements Carriage (lines 19 - 24). Thus *MockProduct* both implement and inherit two other classes, making it cannot be replaced by using Mockito. Then in the test case, the developer creates an instance of *MockProduct* and calls the test target method *getProductPrice* (line 31) with the *MockProduct* instance and a certain delivery distance (i.e. 20.0), the developer then asserts the price with the expected value.
  - Five classes contain complicated design features, such inner classes and collection of classes, such that after the refactoring the test case will become more complicated, and thus it is not beneficial to refactor. For example, as illustrated in Fig. 14b, *ProductService* contains method *decorateProduct* (line 1-6). The developer creates a subclass *MockProduct* for testing this method. To facilitate the logic, the developer further creates an internal interface *ProductDecoratorSupplier* (line 26-28) and provides the implementation of this interface to manipulate how to decorate a given product (line 20) for testing purpose. In the test case, the developer creates the *MockProduct* with a self-defined decorator (line 34) and calls the test target *decorateProduct* (line 36). The developer further asserts the decorated product's price attribute (line 37). In such an example, the developer can extract the interface and still be able to use a mocking framework with a hard-coded Supplier, however, the case would be more complicated. Defining an interanl interface can make the behavior of the *MockProduct* more flexible.
- One class contains syntax that is not captured by Xiao's tool, and thus the refactoring is not successful. This is specific to the tool, and not relevant to the general practice of mocking, thus we do not provide the example here.

The remaining four classes are not (directly) for test dependency isolation. Following, we provide a more detailed discussion of each class. However, due to the unique complexity of these cases, we are not able to provide illustrating examples. However, the readers can explore the original, detailed code with the links if they are interested.

- The first case is MockAsBeanTest from Apache Camel. This class extends ContextTestSupport, which contains basic supporting functions for testing (such as customized assertions and common arrangements). Thus, *ContextTestSupport* serves as a test base class. *MockAsBeanTest* contains "Mock" in its name because it is for testing the production class MockEndpoint, which is a Singleton scope bean.
- The second case is MockImplementation1_EventAnnotationsBase from Apache Cayenne. This class is created to provide input to the target function *DefaultInjector*. It is mocking a random implementation of a method with a special annotation, *@BeforeScopeEnd*,

which is the target of execution to verify the dependency injection mechanism. Thus, this case is a new class created for providing input to a testing scenario.
– The third case is *MockRequestMatcher* from Apache KNOX. It is aggregated in Mockservlet from MockInteraction, which uses inheritance to isolate its parent class *HttpServlet* from the test target. Thus, strictly speaking, *MockRequestMatcher* itself is not for test dependency isolation, but it supports another class for dependency isolation by focusing on customized assertions.
– The last one is AgentDataMock in SkyWalking. This class is not a unit test. It contains a main program that connects several *MockServices* through *ManagementService-BlockingStub*. To our best understanding, this class is a smoke test to verify that the *ManagementServiceBlockingStub* has the ability to connect multiple *MockServices* in a sequence and use *StreamObserver* to monitor the execution for each service. Therefore, it is not for test dependency isolation, but for simulating an integration test scenario in a main function.

We acknowledge that these four cases are not comprehensive in covering all possible scenarios of "mock" without test dependency isolation. But the findings could point to more future research regarding how developers use the concept of "mock" when it is not (directly) relevant to test dependency isolation as what is currently supported by mocking frameworks.

RQ5 Take-aways:

1. Using inheritance for mocking is a common design for test dependency isolation without relying on a mocking framework. But, it is not always possible to use a mocking framework to replace inheritance due to various reasons, such as the limitations of the mocking frameworks and the complexity of the cases. More future research could be beneficial to understand the limitations in different scenarios.
2. Developers may use the concept of "mocking" which is not (directly) relevant to test dependency isolation at all. It requires a more systematic and in-depth investigation to understand the different interpretations of mocking in practice.

5.6 Survey Results

We received a total of 17 responses from the 300 invitations sent out, thus the response rate is 5.7%. Given that the survey is completely anonymous, we are not aware of who are these participants. With this low response rate, we conjecture that this could be due to three reasons: 1) the contact emails mined from git logs are mostly affiliated with "apache.org", which may not be the best contact to reach the developers; 2) the open-source developers are quite dynamic, and thus may not always still active. 3) Our emails may not be successfully delivered to developers because they are recognized as spam. However, Fosnacht's study Fosnacht et al. (2017) suggests that a response rate above 5% should be able to provide similar results as a higher response rate of up to 75%, therefore, we believe this still provides representative results. Following, we will present the survey results. In particular, we will focus on how this validates or provides additional input to each RQ, for the sake of illuminating future research.

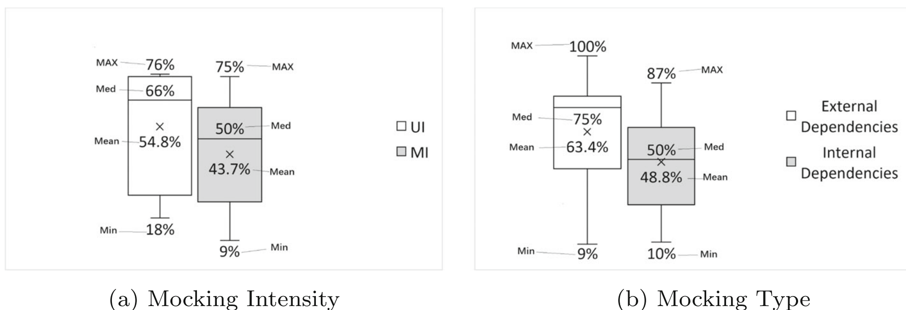
SQ 1-3: Framework Adoption For SQ1 (selection of commonly used frameworks), there were a total of 42 selection counts from the 17 participants, since a participant was allowed to select multiple frameworks. Overall, Mockito, EasyMock, and PowerMock received the highest number of 17, 11, and 6 votes respectively, which add up to 80% of all ballots. In other words, Mockito, EasyMock, and PowerMock were selected by 100%, 64.7%, and

35.3% of the 17 participants. This is consistent with our finding of the top 3 popular mocking frameworks in RQ1.

For SQ2 about the combination of multiple frameworks, 15 (88.2%) participants acknowledge that their projects have used a combination of multiple frameworks. This percentage doubles the 37% of projects with multiple frameworks reported in RQ1. We believe that this discrepancy is likely to be the result of the bias caused by the participants. We conjecture that developers who work on projects with stronger needs for mocking are more likely to participate in our survey, and thus the results tend to be skewed by developers who mock more intensively. We observe the same direction of discrepancy in the SQ4 (mocking intensity), which we will discuss later in more detail.

The most common combination is Mockito as the primary framework, and EasyMock or PowerMock as the secondary framework, which takes a total of 67% selections. This is highly consistent with our findings in RQ1 as well. The participants provided the following explanation for the combinations: 1) Providing a more powerful mock with the combination: *"Mockito is the main mock framework, PowerMock is only used to mock static methods."*, *"PowerMock is more of a club that is used along with EasyMock or Mockito to mock out objects that you otherwise could not (usually involving classloader tricks, like dealing with static objects)"*; 2) Framework migration: *"EasyMock mostly is legacy", "EasyMock has been used from a very beginning, and Mockito wasn't available then, I started using Mockito and replacing EasyMock with Mocktio as it constantly evolves", "EasyMock is like Mockito. Many projects migrate gradually to Mockito."*; and 3) Complementary mocking functions: *"This comparison is a bit apples to oranges Mockito, EasyMock, JMock would be used for mocking Java objects, whereas MockWebserver and WireMock are for stubbing out web servers." and "Mockito is the most preferred one for mocking Java interfaces/classes. WireMock is used for REST APIs mocking."*

SQ 4-5: Mocking Intensity and Type For SQ4, as shown in Fig. 15a, participants report that the Usage Intensity (UI) ranges from 18% to 76%, with an average perception of 55%—meaning that participants believe that on average more than half of the test files in their projects use mocking. Similarly, the Mocking Intensity (MI) ranges from 9% to 75%, with an average of 44%—meaning that participants believe that on average 44% of dependencies are mocked in testing. The perception of the participants is significantly higher than the results we collected from RQ2, where the average UI and MI are 10% and 11% respectively among the projects. We believe this discrepancy is the result of the bias of the participants. As we mentioned earlier, we conjecture that developers who practice mocking intensively are more likely to participate in our survey. Also, the input provided by developers could be



(a) Mocking Intensity (b) Mocking Type

Fig. 15 : Perception of mocking intensity and type

Table 10 : Popularity scores of the most popular APIs

TOP	Mockito	EasyMock
1	when()	expect()
2	thenReturn()	replay()
3	mock()	andReturn()
4	verify()	verify()
5	any()	createMock()
6	doReturn()	createNiceMock()
7	times()	anyTimes()
8	eq()	anyObject()
9	spy()	once()
10	anyString()	atLeastOnce()

skewed subjectively; while the mining results from RQ2 are more objective by its nature. The implication is that mocking could be perceived more intensively than it actually is by developers.

In SQ5, participants evaluated the likelihood of mocking the three different types of dependencies. As shown in Fig. 15b, on average 63.4% of the mocked dependencies are external; while on average 48.8% of the mocked dependencies are internal projects. This is highly consistent with our findings in RQ2.

SQ 6-8: Most and Least Frequent Used APIs: In SQ6, participants were asked to list their top frequently used APIs based on their selected primary framework. Based on the participants' input, we rank the overall popularity of the nominated APIs. Table 10 shows the ranking based on the survey for Mockito and EasyMock respectively. The top 10 popular APIs selected by developers from the survey are the same as the overall top 10 APIs (combining all function groups) we counted in RQ4. This suggests that our findings in RQ4 match the experience of the participants. Note that we did not receive responses for PowerMock, as quoting from a participant again that *"PowerMock is more of a club that is used along with EasyMock or Mockito "* as we mentioned in SQ2.

In SQ7, when we present the 10 least frequently used APIs based on the analysis of RQ4 to participants, the overall selection rate of these APIs is fairly low, which aligns with the message from RQ4. More specifically, for the top 10 least used APIs in Mockito, only 7 APIs were selected mostly once or twice, with an average selection rate of 9.9%. Similarly, the average selection rate for the 10 least frequently used APIs in EasyMock is 9.9% as well. The implication is that on average only 1 in 10 developers may use the 10 least frequently used APIs mined from RQ4. Note that as we discussed earlier, the participants are likely to use mocking more intensively (see SQ4) than average. Thus, the overall developer population is even less likely to use these APIs than reflected by this survey result. Through SQ8, participants explained that they do not know or are not familiar with these APIs (64% participants) or that these APIs are often not useful in their projects (35% participants).

SQ 9-10: Customized Mock and Limitations: For SQ9, 33% of the participants confirmed that they used informal methods other than mocking frameworks. There are two such methods mentioned in the open-ended follow-up question. The first method is to create a completely new class for creating stubs or test doubles for testing. The other method is to define test-only sub-classes, such as implementing interfaces, abstract methods, overriding methods, etc. In our RQ5, the majority (40 out of 44) of the sampled Customized Mocks rely on the second method mentioned by the participants. Furthermore, to our best understanding, one

of the remaining four sample cases, namely *MockImplementational1_EventAnnotatonsBase* should align with the first method of creating completely new classes as test doubles/stubs. The reason is this class is created to provide input to the target function *DefaultInjector*, i.e. a random implementation of a method with a special annotation, @BeforeScopeEnd. For the remaining three cases, each of them associates with the concept of "mock" due to the unique complexity of the testing purposes. This aligns with the input for SQ10 discussed below, that mocking frameworks are often challenged by complicated testing scenarios.

For SQ10, we received three responses that pointing to three different limitations of mocking frameworks: 1) constructing complicated object structures for mocking, such as employing a builder pattern, is challenging using mocking frameworks; 2) mocking static methods and constructors are still difficult with mockito-inline; and 3) Using mocking frameworks is VERY hard to maintain. These call for more future research to address these limitations.

Survey Take-aways:

1. The survey confirmed that Mockito, EasyMock, and PowerMock are the top 3 commonly used frameworks. However, the perception of using multiple frameworks in a project more than doubles the percentage discovered in RQ1 (i.e. 87.5% vs. 37%). The reason behind using multiple frameworks is three-fold: using PowerMock to backup Mockito or EasyMock, framework migration from EasyMock to Mockito, and mocking complementary aspects with different frameworks.
2. The perception of mocking intensity based on the survey is significantly higher than the results in RQ2 based on repository mining. It is possible that participants who practice mocking more intensively than average are more likely to participate in our survey. The mocking type (interval vs. external) from the survey is consistent with the finding of RQ2. The takeaway message is that, although developers use mocking selectively in their projects (based on RQ2), this could drastically vary based on the developers' expertise and experience. Thus, there is a non-technical dimension in terms of when to mock. This is a very rich research question to explore more deeply in future studies.
3. Regarding the most and least frequently used mocking APIs, the survey and RQ4 provide quite consistent observations. This underscores the reliability of our suggested API lists in RQ4 as a "cheat sheet" for practitioners.
4. Developers are aware of the methods behind the Customized Mocks we discovered in RQ5. They pointed to three limitations of existing frameworks that potentially motivate them to the informal mocking methods. These call for more future research to better understand and provide solutions for these challenges.

6 Comparison with Mostafa and Wang's Study

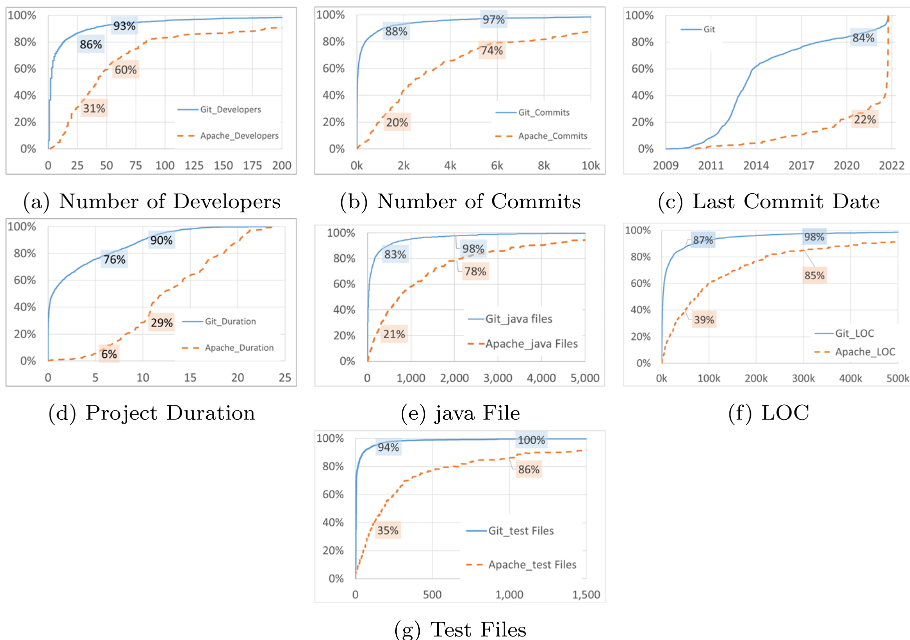
We compared the list of projects included in Mustafa and Wang's study and the Apache projects in our study, there are 8 Apache projects that overlap in both studies, which is trivial compared to the total 246 Apache projects. This is because Mustafa and Wang randomly sampled projects on Github, and many Apache projects were not included, and also many Apache projects were gradually moved to Github after 2014 when Mustafa and Wang sample

their projects. In order to make a meaningful comparison of our study results with that of Mostafa and Wang (2014), we first conduct a comparison of the characteristics of the GitHub projects in Mostafa and Wang (2014) and the Apache projects in our study, from the number of developers, number and time of commits, project duration, project scale, and test code scale, which may have an impact on the adoption of mocking frameworks. Next, we will make a one-on-one comparison of the findings of this paper vs. the paper of Mostafa and Wang (2014).

6.1 Comparison of Apache and GitHub Projects

Number of Developers: Figure 16a compares the number of developers who contributed to the projects on Apache and to the project on Git. We observe that the majority (86%) of GitHub projects have less than 25 developers; while a relatively small portion (31%) of Apache projects have less than 25 developers. In addition, only about 7% (100%-93%) GitHub projects contain more than 50 developers; while close to half (40% = 100%-60%) Apache projects have more than 50 developers. Overall, Apache projects have on average 4 (4=69/17) times more developers than GitHub projects.

Number and Time of Commits: Figure 16b compares the total number of commits in Apache projects and that in GitHub projects. We notice that the majority (88%) of GitHub projects have less than 1k total commits; while only 20% Apache projects have less than 1k commits. About 26% (100%-74%) Apache projects have more than 5k commits, but only 3% (100%-97%) GitHub projects have more than 5k commits. This indicates that Apache projects are going through more code revisions than GitHub projects, which could be the



(a) Number of Developers (b) Number of Commits (c) Last Commit Date

(d) Project Duration (e) java File (f) LOC

(g) Test Files

Fig. 16 : GitHub project vs. Apache project in different dimensions

result of longer history. In addition, Apache projects on average have 6.4 times of commits compared to GitHub projects.

In addition, Fig. 16c compares the date of the most recent commits in Apache projects and that in GitHub projects. In 84% GitHub projects, the most recent commit happened before 2020; while in 78% (100%-22%) Apache projects, the most recent commits happened after 2020. This indicates that most Apache projects still remain activity in the past two years; while most GitHub projects are no longer active in the past two years.

Project Duration: Figure 16d compares the duration (# of years from the start date to the end date) of the projects on Apache and that of the project on Git. We observe that the majority (76%) of GitHub projects have less than five years of history; while only 6% Apache projects have less than five years of history. In addition, the majority (71% = 100%-29%) of Apache projects have more than ten years of history; while only 10% (100%-90%) of GitHub projects have more than ten years of history. Overall, the Apache projects are on average age 4 (4=12/3) times compared to GitHub projects.

Project Scale: Figure 16e compares the number of .Java files in Apache projects and that in GitHub projects. We notice that Apache projects overall contain more Java source file than GitHub projects. More specifically, 83% of the GitHub projects have less than 200 Java files, while only 21% of Apache projects have less than 200 Java files. Likewise, only 2% (2% = 100% - 98%) of GitHub projects had more than 2000 Java files. And 22% (22%=100%-78%) of Apache projects had more than 2000 Java files.

In addition, Fig. 16f compares the lines of code (LOC) of the two communities. Similar to the results of the Java file analysis, the Apache project has more LOCs. More specifically, 87% of the GitHub projects are less than 50KLOC, while only 39% of Apache projects are less than 50 KLOC. In addition, 2% (2%=100%-98%) of GitHub projects are greater than 300 KLOC while 15% (15%=100%-85%) of Apache projects are greater than 300 KLOC.

The above observations regarding number of files and LOC indicate that the Apache projects have larger code size than the GitHub projects. Overall, Apache projects on average have 4.75 times of LOCs compared to GitHub projects. And, Apache projects on average have 7.14 times of .Java files compared to GitHub projects.

Test Code Scale: Figure 16e compares the number of test files in Apache projects and that in GitHub projects. Apache projects have more test files compared to GitHub projects. For example, 93% of the GitHub projects have less than 100 test files; while only 35% of Apache projects have less than 100 test files. Likewise, less than 0.1% of GitHub projects had more than 1000 Test files; but 14% (14%=100%-86%) of Apache projects had more than 1000 test files. Overall, Apache projects on average have 17.76 times of the number of test files compared to GitHub projects.

Summary of Comparison between Apache projects and GitHub projects:

1. Apache projects have on average 4 times more developers than GitHub projects.
2. Apache projects have on average 6.4 times of commits compared to GitHub projects; and most Apache projects still remain active commits, but many GitHub projects became inactive, in the past two years.
3. Apache projects are on average age 4 times compared to GitHub projects;
4. Apache projects are on average 7 times the scale of GitHub projects, measured by the number of Java files in a project.
5. Apache projects on average have 17 times of test files compared to GitHub projects. This indicates that Apache projects probably conduct testing more thoroughly compared to GitHub projects.

Based on the above observation from the comparison, we envision that Apache projects will have higher adoption of mocking frameworks due to their overall larger scale and higher activity level.

6.2 Comparison of Results

Mostafa and Wang's study Mostafa and Wang (2014) focused on overlapping research questions in our study, including RQ1 (overall framework adoption), RQ2 (mocking intensity and type), and RQ4 (frequently used APIs). Later in this section, we will compare how our findings differ and complement (Mostafa and Wang 2014) in these RQs. We would like to also clarify that there are 9 years between the two studies. The highlighted differences of findings could be the result of the time difference.

In addition, we would like to highlight that our study presents more thorough aspects of mocking usage, which are not available in Mostafa and Wang (2014). They include RQ3 (mocking adoption evolution), RQ5 (informal mocking methods), as well as a survey that confirms and deepens our findings in the RQs by taking survey input from developers.

Adoption of Mocking Frameworks (RQ1): In Mostafa and Wang's study Mostafa and Wang (2014), about 23% projects use mocking frameworks. In comparison, 66% of the Apache projects in our study use mocking frameworks. To understand this difference, we provided a quantitative measure of project characteristics, and investigate how they correlate with the adoption rate in projects as shown in RQ 1.2 (Adoption Factor Analysis), which is not available in Mostafa and Wang (2014). 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 is consistent with our finding in RQ-1.2 that larger-scale and newer projects are in greater need of mocking frameworks.

Intensity of Mocking (RQ2): In RQ2, both our study and Mostafa and Wang's study Mostafa and Wang (2014) revealed that mocking is used selectively among test files and to mock selective dependencies. The intensity of usage could drastically depend on the project. It indicates that the use of mocking could depend on the concrete context. Furthermore, 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 dev (2023) and Struts Welcome to the Apache (2023), which frequently mock web services. It remains open research regarding the context of when and where mocking should be used in testing practice.

Frequently Used APIs (RQ4): Mostafa and Wang's study Mostafa and Wang (2014) analyzed the top 10 most popular APIs in EasyMock and Mockito. In comparison, our study conducted a more in-depth analysis of the most frequently used APIs in three aspects. We looked into the three most popular APIs, namely Mockito, EasyMock, and PowerMock. We provided a systematic analysis of the functional categorization of the APIs in each framework by inspecting the official documents. We revealed the most frequently used APIs in each functional group and found that they usually account for the majority of practical use. Thus, our study provides more in-depth empirical observation of how mocking framework APIs are used compared to Mostafa and Wang's study Mostafa and Wang (2014). This provides a more actionable cheat sheet for developers who want to quickly learn how to use the most popular mocking frameworks. Practitioners can further benefit from the common API

sequences for mock object 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.

7 Implications and Future Work

In RQ1, we reported that large-scale and more recent projects are more likely to adopt a mocking framework. However, more in-depth and qualitative investigation in future research could benefit practitioners in guiding the adoption of a proper mocking framework for their projects, such as when and which framework to adopt based on what characteristics of their projects. 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 on Mockito and EasyMock.

In RQ2, we revealed that mocking is practiced overall quite selectively in projects. But in some cases, mock objects are used intensively for some test classes. However, there remains open research regarding the context of when and where mocking should be used in testing practice in a particular project. More empirical knowledge regarding which types of dependencies should or should not be mocked could facilitate the practice of mocking for practitioners.

In RQ3, we revealed different evolution patterns of how the mocking intensity changes over time in projects' history. The usage intensity of mocking functions in different projects shows different trends in the projects' evolution history, i.e. increasing, decreasing, stable, and fluctuating—implying the compound effects of various factors, such as the pace of a project's growth, the available resources, time pressure, and project priority. It calls for more future research to help projects cope with the needs of the mocking with the growth of priorities of the projects.

In RQ4, we revealed "cheat sheets" of the most frequently used APIs based on functions from the top three most popular mocking frameworks. However, mocking APIs are usually used in typical patterns of three steps: 1) create a mock object; 2) stub the mock behavior; and 3) verify the mock object execution. The APIs are not used separately by themselves. It is still open to future research to extract the most common API sequences to facilitate the learning and usage of popular mocking frameworks.

In RQ5, we revealed that the concept of "mock" may not always align with the concept of test dependency isolation. Developers may interpret it differently in different contexts. There is still very limited knowledge of the other types of "mock" besides test dependency isolation, and how we can provide more support on these aspects if existing mocking frameworks are not applicable. In addition, inheritance is a common approach for mocking without using a mocking framework. The adoption of inheritance for mocking could be the result of the limitations of existing mocking frameworks. It is open to future research to systematically investigate such limitations of existing mocking frameworks.

The survey results which represent developers' experience mostly aligned with our study results based on repository mining. One interesting discrepancy is with regard to the intensity of mocking. The perception of the mocking intensity is significantly higher than what is mined from the repository. On the one hand, this could be caused by the bias of the participants that they participated in the survey due to their more intense mocking practice. On the other hand, it is possible that developers may perceive more intensive mocking than reality. The implication is that research about mocking practice contains a non-technical dimension that

involves developers' experience, background, and preference. It remains an open research to investigate the non-technical aspects of mocking practice.

8 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 data here https://github.com/gzhao9/Mock-Apache-Empirical-Study.git.

The key motivation of this study is to understand how mocking frameworks are used to support test dependency isolation in unit test cases. As explained in Section 4.1, we identify all the test files that import JUnit from each project since JUnit is the most commonly used framework for unit testing in Java language. However, we acknowledge that this may be an internal threat to validity because of two reasons. First, this may include false positives since some other types of tests, such as integration, may also import JUnit sometimes. Second, this may include test utility files for creating mock objects and these files may not be test files by themselves. We did not strictly distinguish potential different types of tests and different types of files for supporting tests. However, the findings from this paper still represent valuable empirical knowledge and insights regarding how mocking frameworks are used for test dependency isolation in unit testing. We acknowledge that there are other frameworks like TestNG and Spock. In specific, TestNG is used to support different types of tests, such as integration, and end-to-end; while Spock contains built-in mocking functions compared to Junit. We did not consider these frameworks since they are out of the scope of this study. However, we acknowledge that it is a valuable future direction to study in specific.

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, except RQ3 focusing on project evolution, is based on the most recent version of the code base of the Apache Java projects. Since Apache is quite an active community, its projects are undergoing 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. We also admit that it is a potential threat to validity that when analyzing the number of developers who worked on mocking, we did not accurately track the evolution of all the

mock objects across project history and who edited the mock objects. Instead, our heuristic is that developers who worked on test files with mocking functions should also likely be in charge of maintaining mock objects. The reason is that it requires compiling and configuring each version of a project to accurately identify mock objects and track their evolution. We believe that the heuristic we employed may include some inaccuracy, but it should not impact the overall conclusion of the paper. That is, only a limited and changing number of developers are involved in maintaining mocks in projects' history.

In RQ3.2, we examined the evolution of mocking intensity using the metric of the percentage of files with mocks. We admit that this is a potential threat to validity since the metric may be skewed if a project goes through significant refactoring. For instance, if a test file is refactored and broken down into five files, it is possible that all five files still use mocks; it is also possible that only a subset of them, e.g. one file, still use mocks. In the latter, the mocking intensity metric would become much lower because of this refactoring, but this does not necessarily mean that mocking intensity has truly significantly reduced. We have not accurately examined refactorings in projects and how they may impact the measured intensity.

In RQ5, we search for potential Customized Mock Classes by looking for test files matching the keyword "mock" or "spy" in the names but do not import any mocking framework APIs. Admittedly, on the one hand, it is possible that some Customized Mock Classes may not contain any related keywords; on the other hand, some identified cases are for mocking as intended by a mocking framework. For example, in our manual inspection of the 44 sample cases, 4 cases are not for mocking due to different reasons that are unique to each case. We acknowledge that we did not manually review and inspect all 2,237 cases that match the search criteria in this study. This merits another dedicated study for a more comprehensive and in-depth investigation.

Finally, Apache projects are in different scales and ages, and they cover a variety of problem domains, such as big data, build management, cloud, database, geospatial projects, graphics, etc. Thus, given the diversity of Apache projects, we believe that this study provides representative empirical experience regarding how mocking frameworks are being used in practice by large-scale and long-lived projects. With that being said, we cannot guarantee that the observations we made based on Apache projects will hold and generalize to a set of projects with completely different characteristics. For example, as we explicitly discussed in Section 6, we compared the different observations we made based on Apache projects and Github projects based on Mustafa and Wang's study. How developers use mocking frameworks may be impacted by project characteristics, such as size and domain.

9 Related Work

In the past decade, research related to mocking in software testing has drawn increasing interests. Freeman et al. (2004) was one of the first to propose the basic idea of mocking in unit testing. They contributed a mocking framework named jMock for Java (Freeman et al. 2004).

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. (2007) 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 (2016) 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 and Henriksson (2017) 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 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. (2020) proposed an improved mocking framework MOKA, which is specialized in generating reusable mock objects for mobile apps unit testing. It uses component-based program synthesis to leverage existing test executions to create mock objects automatically. The study shows that this helps developers to repair the tests that have external data dependency. Another study of Fazzini et al. (2022) focused on test doubles used in Android apps, examining their creation and usage in 1,006 apps. The study identified commonly used frameworks and methods, analyzing the specifics of 2,365 test doubles across 10 apps with the highest usage. The paper concludes that Android's test doubling practices diverge significantly from traditional Java, potentially leading to test smells and errors.

With the prevalent usage of mocking framework, researchers also started to focus on improving the education and design of mock objects. Nandigam et al. (2009) 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 and Marshall (2016) 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 and Hora (2020) 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. (2009) 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. (2017) incorporated a mocking framework to automated unit test generation. Their study confirmed the anticipated improvements in code coverage and bug detection. Zhu et al. (2020) 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. (2021) 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.

Studies that are closest to ours include (Mostafa and Wang 2014; Spadini et al. 2017, 2019), which all presented empirical studies regarding how mocking is practiced. Of particular note,

our study is most similar to study (Mostafa and Wang 2014) in that both investigated how mocking frameworks are used in open-source communities. As presented in Section 6, we provided a very detailed comparison of our study and Mostafa and Wang (2014), highlighting our unique contributions in that 1) we focused on a different community which provides findings that complement (Mostafa and Wang 2014); and 2) we conducted more thorough investigations regarding factors that impact framework adoption, the evolution of mocking framework adoption, informal mocking methods other than mocking frameworks, as well as a survey focuses on developers' perception of mocking framework usage, which is not available in Mostafa and Wang (2014).

Furthermore, Spadini et al. (2017, 2019) investigated the usage of Mockito and the evolving process of mock objects in three OSS projects and one industrial system. The key focus of their studies Spadini et al. (2017, 2019) is on gaining an in-depth understanding of what types of objects are more likely to be mocked. 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. In comparison, our study presented a more comprehensive investigation of mocking practiced in 193 Apache projects. Our study answers a broad range of research questions that cover: 1) the overall adoption of different mocking frameworks across the entire community, how different project factors impact the adoption of mocking frameworks; 2) the intensity of mocking in projects and type of objects being mocked, which is most relevant to Spadini et al. (2017, 2019), but we based on a much larger dataset (i.e. 193 projects vs. 4 projects); 3) the overall evolution trends of mocking framework adoption in projects in the community; 4) the most frequently used mocking APIs in the top 3 most popular frameworks (Mocito, EasyMock, and PowerMock); 5) informal methods for mocking other than using a framework; and lastly, 6) a survey that involves developers to confirm and deepen our understanding of the above aspects. Therefore, we believe that our study provides significant new and complementary knowledge that is not available in existing, similar empirical studies Mostafa and Wang (2014); Spadini et al. (2017, 2019).

Finally, while the popular mocking frameworks provide comprehensive tutorials to teach practitioners how to use them, practitioners still could benefit greatly from our study in the following aspects. First, the tutorials provide basic concepts of when and what to mock; while our study shares rich empirical experience from hundreds of projects, with quantitative measures regarding what project factors contribute to the adoption of mocking frameworks, the different types of objects frequently being mocked, as well as informal methods for mocking other than using a framework. Practitioners could gain more practical experience from our study. Second, the tutorials provide very detailed descriptions of each single mocking API. As presented earlier, there are a total of 317 APIs in Mockito, 278 APIs in EasyMock, and 311 APIs in PowerMock, which are the three most popular mocking frameworks. But only 109 (34.5%), 68 (24.5%), and 75 (24.1%) APIs in Mocito, EasyMock, and PowerMock, respectively, are actually used by Apache projects. In our RQ4, we derived key function categories provided by the APIs based on the tutorial and mined the top 5 frequently used APIs in each functional group. This short-list of APIs takes the majority (78% to 100%) of usage in Apache projects. Therefore, this provides a quick "cheat sheet" to potentially help practitioners accelerate the learning process.

10 Conclusion

This study contributes an in-depth empirical study to reveal whether and how mocking frameworks are used in Apache projects. This study contributes valuable findings and implications to practitioners who are interested in learning and using mocking frameworks, and to those who are interested in related future research directions. The key findings and contributions of this study include the following.

First, mocking frameworks are widely used in 66% of Apache projects, with higher adoption among larger-scale and newer projects. Thus, developers who work on real-life, modern software projects should generally be prepared to use mocking frameworks. Practitioners and educators should prioritize Mockito, EasyMock, and PowerMock since they are the top three most popular mocking frameworks.

Second, mocking could be practiced selectively in general, but also intensively in cases—varying based on specific contexts. More research is needed to investigate when and what to mock in software testing, and by whom.

We also revealed that projects could show different evolution patterns of framework adoption—increasing, decreasing, stable, and fluctuating. It points to a future direction to gain a deeper understanding of what impacts the evolution of mocking framework adoption, and how to help projects in need to keep a healthy pace in using mocking that match the growth of the projects.

Furthermore, given the hundreds of total APIs in a mocking framework, developers usually only need to focus on the top five APIs for creating mock objects, stubbing mock behaviors, and verifying mock execution, to handle the majority of usage scenarios in practice. This study provides quick cheat sheets of those APIs for Mockito, EasyMock, and PowerMock.

Despite the power functions of mocking frameworks, we found that inheritance is a common informal method of enabling test dependency isolation, which aligns with the goal of mocking frameworks. But it is not always possible or beneficial to replace inheritance using a mocking framework. Also, developers may have other interpretations of mocking, other than test dependency isolation supported by existing mocking frameworks. It calls for more future research to understand the current limitations of mocking frameworks to better support various, complicated mocking needs in practice.

Finally, the developer survey provided confirmation and additional insights into the findings of our study, which could illuminate various potential future research questions related to mocking usage in practice.

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

Data Availability We have published all the raw and generated data on GitHub https://github.com/gzhao9/Mock-Apache-Empirical-Study.git. The data is organized based on the structure of the research questions.

Declarations

Conflicts of interest The authors declared that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is

not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Apache PDFBox | a java PDF library. https://pdfbox.apache.org/

Apache software foundation projects list. https://projects.apache.org/projects.html. Accessed 8 May 2023

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

Barker D (2016) Web content management: systems, features, and best practices. " O'Reilly Media, Inc."

Bertolino A (2007) Software testing research: achievements, challenges, dreams. In: Future of software engineering (FOSE'07), IEEE, pp 85–103. https://doi.org/10.1109/FOSE.2007.25

Briney K (2015) Data Management for Researchers: Organize, maintain and share your data for research success. Pelagic Publishing Ltd

Buyya R, Dastjerdi AV (2016) Internet of Things: Principles and paradigms. Elsevier

Crowston K, Howison J (2006) Assessing the health of open source communities. Computer 39(5):89–91

Daka E, Fraser G (2014) A survey on unit testing practices and problems. In: 2014 IEEE 25th International symposium on software reliability engineering, IEEE, pp 201–211. https://doi.org/10.1109/ISSRE.2014.11

dev: Apache wink – index. https://wink.apache.org/

Duenas JC, Cuadrado F, Santillán M, Ruiz JL et al (2007) Apache and eclipse: comparing open source project incubators. IEEE Soft 24(6):90–98

EasyMock. https://easymock.org/

Fazzini M, Choi C, Copia JM, Lee G, Kakehi Y, Gorla A, Orso A (2022) Use of test doubles in android testing: an in-depth investigation. In: Proceedings of the 44th international conference on software engineering, pp 2266–2278

Fazzini M, Gorla A, Orso A (2020) A framework for automated test mocking of mobile apps. In: 2020 35th IEEE/ACM International conference on automated software engineering (ASE), IEEE, pp 1204–1208

Fosnacht K, Sarraf S, Howe E, Peck LK (2017) How important are high response rates for college surveys? Rev Higher Educ 40(2):245–265

Freeman S, Mackinnon T, Pryce N, Walnes J (2004) 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, pp 4–5

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

Garousi V, Zhi J (2013) A survey of software testing practices in Canada. J Syst Softw 86(5):1354–1376. https://doi.org/10.1016/j.jss.2012.12.051. https://www.sciencedirect.com/science/article/pii/S0164121212003561

Henderson F (2017) Software engineering at google. arXiv:1702.01715

Hunt A, Thomas D (2004) Pragmatic unit testing in c# with nunit. Pragmatic Programmers

Ieee standard glossary of software engineering terminology (1990) IEEE Std 610(12–1990):1–84. https://doi.org/10.1109/IEEESTD.1990.101064

JUnit 5. https://junit.org/junit5/

Kaner C, Falk J, Nguyen HQ (1999) Testing computer software. John Wiley & Sons

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

Kim SS (2016) Mocking embedded hardware for software validation. Ph.D. thesis

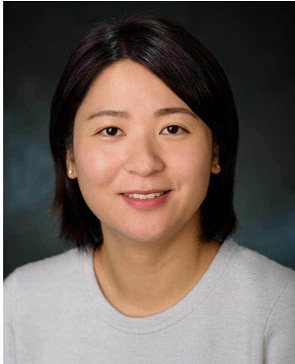
List of releases for the project. https://github.com/gzhao9/Mock-Apache-Empirical-Study/blob/main/RQ1/project%20tags_info.csv

Marri MR, Xie T, Tillmann N, De Halleux J, Schulte W (2009) An empirical study of testing file-system-dependent software with mock objects. In: 2009 ICSE Workshop on automation of software test, IEEE, pp 149–153. https://doi.org/10.1007/s10664-018-9663-0

Maven Repository: org.springframework » spring-mock. https://mvnrepository.com/artifact/org.springframework/spring-mock

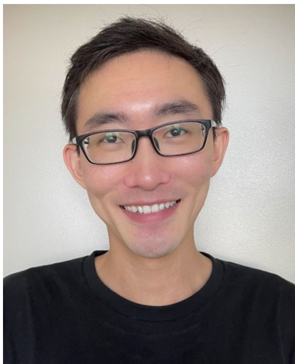
Mockito release notes. https://code.google.com/archive/p/mockito/wikis/ReleaseNotes.wiki. Accessed 8 May 2023
mockito. https://site.mockito.org/
Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and mozilla. ACM Trans Softw Eng Methodol (TOSEM) 11(3):309–346
Mockus A, Fielding RT, Herbsleb J (2000) A case study of open source software development: the apache server. In: Proceedings of the 22nd international conference on software engineering, pp 263–272
moq. https://github.com/moq/moq4
Mostafa S, Wang X (2014) An empirical study on the usage of mocking frameworks in software testing. In: 2014 14th international conference on quality software, IEEE, pp 127–132. https://doi.org/10.1109/QSIC.2014.19
Myers GJ, Badgett T, Thomas TM, Sandler C (2004) The art of software testing, vol 2. Wiley Online Library. https://doi.org/10.1002/9781119202486
Nandigam J, Gudivada VN, Hamou-Lhadj A, Tao Y (2009) Interface-based object-oriented design with mock objects. In: 2009 Sixth international conference on information technology: new generations, IEEE, pp 713–718. https://doi.org/10.1109/ITNG.2009.268
NMock: A Dynamic Mock Object Library for .NET. https://nmock.sourceforge.net/
Pereira G, Hora A (2020) Assessing mock classes: an empirical study. In: 2020 IEEE International conference on software maintenance and evolution (ICSME), IEEE, pp 453–463. https://doi.org/10.1109/ICSME46990.2020.00050
PowerMock framework site. https://powermock.github.io/
PowerMock framework site. https://powermock.github.io/
Projects by category in apache software foundation. https://projects.apache.org/projects.html?category
Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: a case study of the apache server. In: Proceedings of the 30th international conference on Software engineering, pp 541–550
Runeson P (2006) A survey of unit testing practices. IEEE Softw 23(4):22–29. https://doi.org/10.1109/MS.2006.91
Severance C (2012) The apache software foundation: Brian behlendorf. Computer 45(10):8–9
Solms F, Marshall L (2016) Contract-based mocking for services-oriented development. In: Proceedings of the annual conference of the south african institute of computer scientists and information technologists, pp 1–8
Spadini D, Aniche M, Bruntink M, Bacchelli A (2019) Mock objects for testing java systems. Empirical Softw Eng 24(3):1461–1498. https://doi.org/10.1007/s10664-018-9663-0
Spadini D, Aniche M, Bruntink M, Bacchelli A (2017) 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, pp 402–412. https://doi.org/10.1109/MSR.2017.61
Svensgård S, Henriksson J (2017) Mocking saas cloud for testing
Taneja K, Zhang Y, Xie T (2010) Moda: automated test generation for database applications via mock objects. In: Proceedings of the IEEE/ACM international conference on Automated software engineering, pp 289–292
Unittest.mock - mock object library. https://docs.python.org/3/library/unittest.mock.html#module-unittest.mock
Wang X (2021) Understanding and facilitating the usage of mocking frameworks for test dependency isolation. Ph.D. thesis, Stevens Institute of Technology
Wang X, Xiao L, Yu T, Woepse A, Wong S (2022) From inheritance to mockito: An automatic refactoring approach. IEEE Trans Softw Eng 1–23. https://doi.org/10.1109/TSE.2022.3231850
Wang X, Xiao L, Yu T, Woepse A, Wong S (2021) 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, pp 540–552
Web E (2013) Eclipse JDT™(Java development tools). https://projects.eclipse.org/projects/eclipse.jdt
Weiss M, Moroiu G, Zhao P (2006) Evolution of open source communities. In: Open source systems: IFIP working group 2.13 foundation on open source software, June 8–10, 2006, Como, Italy 2, Springer, pp 21–32
Welcome to the Apache Struts project. https://struts.apache.org/
Why is it so bad to mock classes? — stackoverflow.com. https://stackoverflow.com/questions/1595166/why-is-it-so-bad-to-mock-classes. Accessed 19 Jul 2023
Zhu H, Wei L, Wen M, Liu Y, Cheung SC, Sheng Q, Zhou C (2020) Mocksniffer: characterizing and recommending mocking decisions for unit tests. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, pp 436–447

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

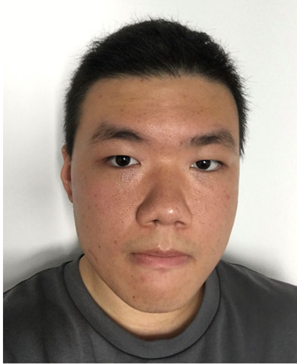


Lu Xiao is an Assistant Professor in the School of Systems and Enterprises at Stevens Institute of Technology. Her research interests lie in the broad area of software engineering, particularly in software architecture, software economics, cost estimation, and software ecosystems. She is an awardee of NSF CAREER project in 2021. She has published her work in different conferences and journals, including TSE, ICSE, FSE, and ICSA, etc.. She completed her PhD in Computer Science at Drexel University in 2016. She received the first-place prize at the ACM Student Research Competition in 2015.

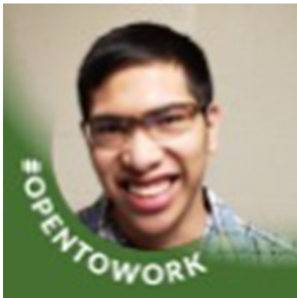
Gengwu Zhao is a Ph.D. candidate advised by Dr. Lu Xiao at the School of Systems and Enterprises, Stevens Institute of Technology. His research interest is software testing.



Xiao Wang received the PhD degree in system engineering with a concentration on software engineering from the Stevens Institute of Technology, in 2022, advised by Lu Xiao. He is a Senior Software Development Engineer with Amazon. His research interests lie in software architecture, software refactoring, software testing and cyber-physical systems. He published his work in different journals and conferences, including IEEE Transactions on Software Engineering, ICSE, FES Journal of Engineering Sciences, and International Chinese Statistical Association.

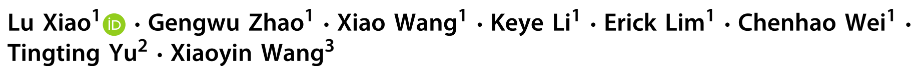


Keye Li is a Master's student at the Computer Science & Engineering department at University of California, San Diego. He obtained his Bachelor's degree in Software Engineering at Stevens Institute of Technology in 2022. His current research interests lie in distributed systems.



Erick Lim received his Bachelor's degree in Software Engineering in 2022 and a Masters' Degree in Computer Science in 2023, both from Stevens Institute of Technology.

Authors and Affiliations



Lu Xiao[1] · **Gengwu Zhao**[1] · **Xiao Wang**[1] · **Keye Li**[1] · **Erick Lim**[1] · **Chenhao Wei**[1] · **Tingting Yu**[2] · **Xiaoyin Wang**[3]

Gengwu Zhao
gzhao9@stevens.edu

Xiao Wang
xwang97@stevens.edu

Chenhao Wei
cwei7@stevens.edu

Tingting Yu
tingting.yu@uc.edu

Xiaoyin Wang
xiaoyin.wang@utsa.edu

1 School of Systems and Enterprises, Stevens Institute of Technology, Castle Point Terrace, Hoboken NJ 07030, USA

2 University of Cincinnati, 2600 Clifton Ave, Cincinnati OH 45221, USA

3 University of Texas at San Antonio, 1 UTSA Circle, San Antonio TX 78249, USA