Understanding and Characterizing Mock Assertions in Unit Tests

HENGCHENG ZHU, The Hong Kong University of Science and Technology, China
VALERIO TERRAGNI, The University of Auckland, New Zealand
LILI WEI, McGill University, Canada
SHING-CHI CHEUNG*, The Hong Kong University of Science and Technology, China
JIARONG WU, The Hong Kong University of Science and Technology, China
YEPANG LIU, Southern University of Science and Technology, China

Mock assertions provide developers with a powerful means to validate program behaviors that are unobservable to test assertions. Despite their significance, they are rarely considered by automated test generation techniques. Effective generation of mock assertions requires understanding how they are used in practice. Although previous studies highlighted the importance of mock assertions, none provide insight into their usages. To bridge this gap, we conducted the first empirical study on mock assertions, examining their adoption, the characteristics of the verified method invocations, and their effectiveness in fault detection. Our analysis of 4,652 test cases from 11 popular Java projects reveals that mock assertions are mostly applied to validating specific kinds of method calls, such as those interacting with external resources and those reflecting whether a certain code path was traversed in systems under test. Additionally, we find that mock assertions complement traditional test assertions by ensuring the desired side effects have been produced, validating control flow logic, and checking internal computation results. Our findings contribute to a better understanding of mock assertion usages and provide a foundation for future related research such as automated test generation that support mock assertions.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **General and reference → Empirical studies**.

Additional Key Words and Phrases: Software Testing, Test Doubles, Mocking

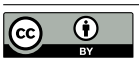
ACM Reference Format:
Hengcheng Zhu, Valerio Terragni, Lili Wei, Shing-Chi Cheung, Jiarong Wu, and Yepang Liu. 2025. Understanding and Characterizing Mock Assertions in Unit Tests. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE026 (July 2025), 22 pages. https://doi.org/10.1145/3715741

1 Introduction

Unit testing aims to verify the correctness of the system under test (SUT) in isolation. However, in typical software, an SUT needs to interact with other components to fulfill its intended functionality. When testing a SUT, developers often substitute its dependent components with test doubles [17].

*Shing-Chi Cheung is the corresponding author.

Authors' Contact Information: Hengcheng Zhu, hzhuaq@connect.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Valerio Terragni, v.terragni@auckland.ac.nz, The University of Auckland, Auckland, New Zealand; Lili Wei, lili.wei@mcgill.ca, McGill University, Montréal, Canada; Shing-Chi Cheung, scc@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Jiarong Wu, jwubf@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Yepang Liu, liuyp1@sustech.edu.cn, Department of Computer Science and Engineering, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China.



ACM 2994-970X/2025/7-ARTFSE026
https://doi.org/10.1145/3715741

Test doubles are objects that simulate the behaviors of such dependent components in controlled ways. An important role played by test doubles is mocking, where they record the method calls (along with their arguments) made to them, allowing developers to write *mock assertions* to verify if the expected method invocations occurred during testing. Mock assertions can validate program behaviors that are unobservable to *test assertions* (e.g., those written using JUnit or AssertJ). Consider the example in Figure 1, where the SUT saves data in a database by calling a method of a database access object (DAO). Since the data are passed to the method call to DAOs without changing any variables or return values that are accessible in the test case, the data value is unobservable to test assertions. Therefore, test assertions cannot check whether the data written to the database is null. Fortunately, developers can leverage mock assertions to verify the recorded method calls to a mock DAO. By checking the arguments passed to the method call that sends out the data, developers can validate the correctness of database access.

Despite their importance in assuring software quality, mock assertions are rarely considered in existing automated test generation techniques. The assertions constructed by test generators (e.g., Randoop [20], EvoSuite [12]) cannot predicate program behaviors that are unobservable to the test assertions and thus have weak test oracles [27]. Techniques like AgitarOne [1] and Rick [31] do generate mock assertions. However, they adopt a brute-force strategy to generate mock assertions for all method invocations of test doubles, resulting in aggressive mocking [27] (i.e., the generated tests contain excessive mock assertions that overfit the current implementation), making the tests inflexible to changes in the SUT. Shamshiri et al. [27] reported that 31% of the tests generated by AgitarOne raise false alarms due to aggressive mocking. Indeed, during our study, we found such a brute-force strategy misaligned with developers' practice. In fact, the aggressive mocking issue is an obstacle for EvoSuite to adopt mock assertions [5] as there is no effective mechanism to identify which interactions between the SUT and test doubles should be verified [10].

Generating effective mock assertions requires understanding their usage in practice. Although several studies emphasize the importance of mock assertions [10, 28, 29, 35, 39], none provides such insights. To bridge this gap, we conducted the first empirical study on the usage of mock assertions. Our study distills insights and empirical evidence for future research exploring the generation of appropriate mock assertions. We also provide guidance for developers to write better mock assertions. Specifically, we investigated: (1) the frequency of using mock assertions when using test doubles (2) the characteristics of method invocations that are verified by mock assertions, and (3) how mock assertions complement test assertions in fault detection.

In our study, we analyzed the usage of mock assertions in 4,652 test cases from 11 large-scale, popular Java projects that use Mockito[1] [8]. We adopted open coding [15] to identify the common characteristics of the method invocations that are verified by mock assertions. In addition, we performed mutation analysis [13] to investigate how mock assertions complement test assertions in fault detection. We had several interesting findings in our study. For example, although mock assertions are used in 41% of the test cases using test doubles, we found that verifying everything with mock assertions is not the state of the practice. Instead, only 9% of method invocations are verified (Section 4). The contrast between the frequent adoption and the low verification ratio motivated us to investigate the characteristics of the method calls verified by mock assertions. During our investigation, we identified three categories of methods whose invocations are commonly verified by developers (Section 5) and two types of common interactions between the SUT and the verified method invocations (Section 6). Finally, we find that mock assertions complement test assertions by ensuring that the desired side effects have been produced, validating control flow logic, and

[1]Mockito is the most popular (used in over 80% of the projects) mocking framework in Java [10, 29]

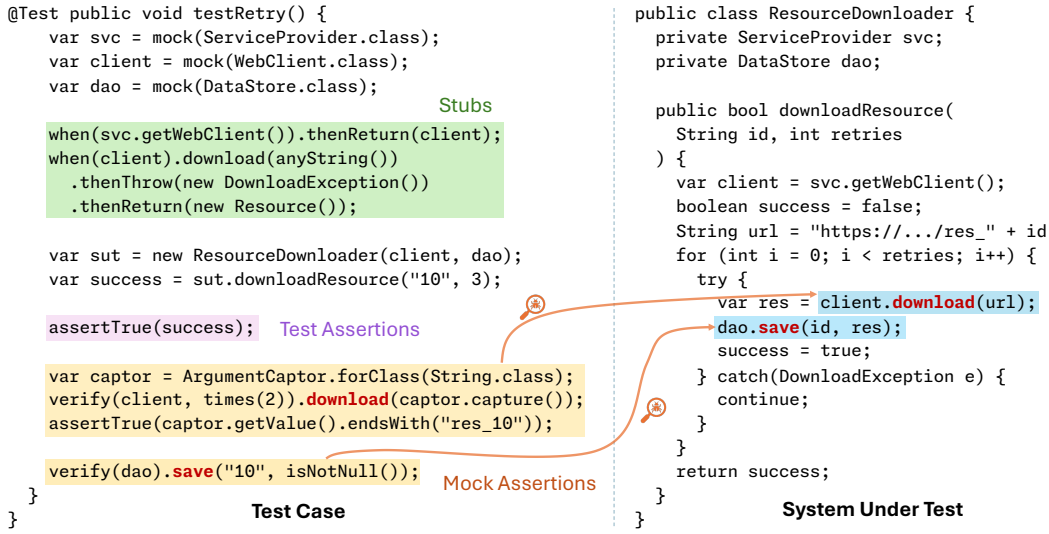


Fig. 1. An Illustration of Mock Assertions and Test Assertions in Unit Testing. Mock assertions are leveraged to check if the URL used for downloading is correct and whether the resource has been saved to the database.

checking internal computation results (Section 7). This finding aligns with our identified characteristics of method invocations verified by mock assertions. Currently, the identification of such characteristics still relies on manual efforts. Future research can explore automated mechanisms to pinpoint critical method invocations during test execution and generate mock assertions to verify them accordingly.

In essence, we have made the following contributions in this paper:

- To the best of our knowledge, we are the first to investigate the characteristics of mock assertion usage in practice.
- We identified three major categories of methods and two major categories of interactions that are often verified by mock assertions. Our findings can shed light on the state of the practice of using mock assertions and provide guidance for both future researchers and developers.
- We investigated the fault detection capabilities of mock assertions and test assertions. We found mock assertions complement test assertions by ensuring the desired side effects have been produced, validating control flow logic, and checking internal computation results.
- We constructed a dataset of method calls that are verified by developers with mock assertions. We release it with our experimental data to facilitate future research endeavors. Our research artifact is available at https://doi.org/10.5281/zenodo.14695509 [37].

2 Background

Test doubles and mock assertions are crucial components of software testing to deal with test dependencies. In this section, we provide an overview of the roles played by test doubles and demonstrate the utilization of mock assertions through an illustrative example.

2.1 Test Doubles in Unit Testing

In software testing, test doubles [10, 17] are simulated objects that represent the actual dependencies during testing. They can be designed to mimic the behavior of real objects in controlled

environments or utilized to verify interactions between various system components without directly involving the actual dependencies.

Test doubles, often created using mocking frameworks like Mockito in Java, can be configured to return specific values, simulate error conditions, or verify that certain methods are called with the expected arguments. Depending on their usage, test doubles can play the following roles.

- **Mock:** Mock objects are configured with expectations about the interactions they will have with the SUT. They record the method calls made to them and enable developers to verify if the anticipated interactions have occurred. Also, the arguments used in these methods calls are also recorded for further checking. For example, the test double `dao` in Figure 1 is a mock.
- **Spy:** Spy objects are a specialized type of mock object that wraps a real object. In addition to recording method calls, they also call the real methods in the wrapped objects.
- **Stub:** Stub objects provide predefined responses to method calls. They return predefined values and are used to simulate specific behaviors of dependencies. For example, the test double `svc` in Figure 1 is a stub.
- **Fake:** Fake objects are more sophisticated versions of stubs. Instead of returning predefined values, fakes offer a lightweight implementation of the actual component solely for testing purposes, such as simulating a database by storing data in an array.
- **Dummy:** Dummies are objects with specific types or implementing certain interfaces. They lack any logic and are merely used to fulfill method signatures during testing. Neither the test nor the SUT interacts with them.

In practice, test doubles created with common frameworks like Mockito can play one or more of these roles. For example, in Figure 1, the test double `client` is used as both stub and mock. In this paper, we focus on mock assertions, which are used with the test doubles of the role mock or spy.

2.2 Mock Assertions vs. Test Assertions

In unit testing using test doubles, there are two types of assertions that can be used to predicate on the program behaviors to assure the correctness of SUT.

- **Test Assertions.** Test assertions are executable boolean expressions in the test that predicate the values of variables. The variables predicated by test assertions are either the output returned from the SUT or the states mutated by the SUT. In practice, developers use common testing frameworks such as JUnit and AssertJ to write test assertions.
- **Mock Assertions.** Mock assertions are assertions that predicate on the recorded method invocations on mock or spy objects. They enable developers to check if the interactions between the SUT and test doubles have occurred in expected ways. Specifically, developers can check whether a certain method of a test double has been invoked, the number of invocations, and whether the arguments used for the method invocations match the expected values. In addition, mock assertions allow developers to capture the value of arguments and further predicate them with JUnit assertions.

Mock assertions complement test assertions by checking the program behaviors that cannot be observed by test assertions. Figure 1 shows an example of unit testing with mock assertions using the Mockito framework. The SUT `ResourceDownloader` downloads a resource using a `WebClient` and stores the result in the database with `DataStore`. Since the download may fail, callers of the method `downloadResource` can specify the number of maximum retries via parameter `retries`. The test case `testRetry` validates the retry logic. Before calling the method `downloadResource`, a mock `WebClient` is configured to simulate the scenario where the method `download` will fail for the first invocation and succeed in the second try. In this case, in addition to returning `true` to indicate a successful operation, the method `downloadResource` should try to download exactly twice and

finally store the resource using `DataStore`. Since the download operation should be successful given such a scenario, developers checked the return value of `downloadResource` with a test assertion.

However, this test assertion alone cannot ensure the correctness of the implementation as several key operations are unchecked. First, the resource should be downloaded exactly twice since the first download failed and the second download was successful. Second, the correct URL should be used for the download. Third, the downloaded resource should be saved into the database. These operations are unobservable in the test since the relevant data is not returned by `downloadResources` or accessible via any getters. Therefore, they cannot be checked by test assertions. Fortunately, these operations manifest as the interactions between the SUT and the test dependencies. In this example, the download and database access are done by calling the methods in `WebClient` and `DataStore`. Therefore, developers use mock assertions to ensure these expected behaviors have occurred. To validate the retry logic, developers use a mock assertion to check if `client.download()` was called twice. The argument passed to the method `download` is also captured with an `ArgumentCaptor`, and the captured value is further checked with a JUnit assertion. Although such an assert statement is similar to test assertions, it is considered as part of the mock assertions since it predicates the value captured by a mocked method call. In addition, to ensure the resource is stored in the database, developers use another mock assertion to check if `dao.save()` was invoked as expected.

In general, mock assertions enable developers to predicate on the behaviors that are not directly observable from the test. This is done by checking the recorded method calls (and their arguments) on mock objects. Mock assertions complement test assertions and make the test stronger.

3 Empirical Study Design

In this section, we outline the design of our empirical study design. Specifically, we outline the research questions and the rationale behind them, and we present our data collection process.

3.1 Research Questions

The goal of this paper is to guide developers in effectively using mock assertions and offer insights to future researchers for developing automated techniques to generate and maintain mock assertions. To achieve our goal, we investigated the following four research questions (RQs):

- **RQ1: Adoption.** *How frequently do developers use mock assertions when using test doubles?* This RQ focuses on understanding the frequency of mock assertion adoption among developers, providing insights into the prevalence of this assertion type in practice.
- **RQ2: Method Types.** *What kinds of method calls are verified by mock assertions?* This RQ explores the types of methods that developers typically verify their invocations using mock assertions, offering understandings of the specific areas in which mock assertions are used.
- **RQ3: Interactions.** *How do the SUTs interact with the methods verified by mock assertions?* In practice, developers use mock assertions to verify interactions between the SUT and the test dependencies [10, 29, 39]. Therefore, we seek to investigate the characteristics of these interactions, aiming to gain insights into the interactions that are of particular interest to developers when using mock assertions.
- **RQ4: Fault Detection.** *How do mock assertions complement test assertions in fault detection?* This research question examines the role of mock assertions in detecting potential faults. We specifically focus on test cases where both assertion types are employed. Our objective is to shed light on the areas where mock assertions can complement test assertions, thereby contributing to the assurance of software reliability.

3.2 Data Collection

To support the investigation of the four research questions, we constructed a dataset of unit tests that interact with test doubles. We detail our data collection process as follows.

Project Selection. We used the query *language:java archive:false pushed:>2024-01-01* to search for repositories on GitHub to identify actively maintained Java projects that have been updated since 2024. The search was done in March 2024 and it returned over 2 million projects. We sorted the search results by the number of stars, which is an indicator of popularity. Popular projects are widely used and are more likely to attract experienced contributors following best practices (e.g., appropriate use of mock assertions). Prioritizing popular projects for our study allows us to generate better findings for developers and future researchers. Beyond popularity, we applied the following criteria to ensure subject quality:

(1) The project contains at least 10k lines of Java code, filtering out smaller toy projects.
(2) We manually reviewed the README file to make sure that the project is not a tutorial or an Android project, as such projects fall outside the scope of this study.
(3) It declares Mockito as a dependency in the Maven POM file or Gradle build scripts because our data extraction tool is implemented based on Mockito, the most widely used mocking framework in Java [10, 19].

We cloned the repositories of the most stared 20 projects meeting the above criteria. We followed their documentation to build their latest release version and run the tests. We discarded a project if we were unable to build it or run the test. Finally, we got a list of 11 projects, as detailed in Table 1. These projects are large-scale, sophisticated, and span major application domains of the Java programming languages (e.g., bug data, web, database), which bolsters our findings in generalizing to these domains.

Unit Tests Identification. In this paper, we focus on unit tests, since test doubles are used primarily in unit tests to isolate the SUT from its dependencies [29]. To identify unit tests in these projects, we adopted an approach similar to that used in recent studies [35, 39]. Specifically, we identify the methods annotated with the JUnit `@Test` annotation as test cases. To ensure that a test is indeed a unit test, we follow the pattern used by the JUnit test runner. We infer the name of the SUT by stripping the prefix or suffix `Test` from the name of the test class and check if there exists such a class in the same package [32, 39]. Then we ran the tests with an instrumented Mockito to check if they interact with the test doubles (i.e., invoke at least one method of test doubles). We aim to exclude test cases that use test doubles as dummies since they are out of the scope of this paper. As shown in Table 1, we identified 4,652 test cases that interact with test doubles.

Method Call Extraction. To construct a dataset of method calls that are verified by developers, we ran all the identified test cases for data extraction. We opted for a dynamic analysis approach due to its ability to capture runtime interactions between the SUT and mock objects during test execution. Dynamic analysis allows us to observe actual method calls and their contexts, which enhances the accuracy of our findings regarding mock assertion usage. Similar approaches have been adopted by recent studies [10, 39] for data collection.

During test execution, we used an instrumented Mockito to collect the runtime interactions between the SUT and test doubles. Specifically, we attached a debugger to the test runner and set a breakpoint in `MockitoCore`, which is used by Mockito internally to generate test doubles. When the breakpoint is hit, we mutated the `MockSettings` object to inject a `InvocationListener` to track method calls to the test double being created. After the test execution, we used the API `Mockito.mockingDetails` to inspect the interactions between the SUT and the test doubles, and

Table 1. Overview of Studied Projects and Test Cases

Project	Application Domain	LoC in Java	Unit Test Cases			Test Doubles Created
			Total	w/ TD[1]	w/ MA[2]	
Camel	Application integration	1,874k	12,535	416	180	1,159
CXF	Web services	837k	5,592	446	94	1,328
Dubbo	RPC framework	286k	3,481	207	52	404
Hadoop	Big data	2,726k	14,013	995	375	3,221
Hazelcast	In-memory data grid	1,414k	22,482	185	64	389
Kafka	Event streaming	190k	7,644	575	241	1,188
MyBatis3	Database ORM	64k	1,664	301	185	365
Neo4j	Graph database	883k	5,457	777	435	2,179
Ozone	Object storage	1,165k	3,031	165	67	482
Spring Boot	Web framework	428k	4,568	512	185	920
Storm	Real-time processing	337k	594	73	39	263
Total		10,204k	81,061	4,652	1,917	11,898

1: With Test Doubles – Test cases invoking at least one method of a test double.
2: With Mock Assertions – Test cases containing at least one mock assertion.

identify the method calls verified by mock assertions. We disabled garbage collection for the test doubles to enable after-test inspection.

The information collected during test execution is gathered to form our dataset. Each entry of our dataset corresponds to a method invocation and it contains the following information: (1) the type and object ID of the test double, (2) the signature of the method call, (3) the stack trace of the method call, (4) a label indicating if developers stubbed a custom return value to the method call, and (5) a label indicating if the method call is verified. We extracted stack traces at the injected breakpoints and retrieved stub/verify labels from Mockito API `MockingDetails`. We eliminated duplicates from the dataset by removing the method calls in the same test case with identical method signatures, mock object IDs, and stack traces, aiming to prevent skewing the dataset by repeated method calls, such as those within loops.

Table 2 presents statistical information of the method invocations to test doubles. During test execution, the SUTs frequently interact with the mock objects. The 4,652 test cases made 40,639 method invocations to the mock objects. Such a frequency aligns with a recent study [10] conducted on Android applications. Among these method invocations, developers specified a custom return value for 24,928 (61%) of them. In addition, developers verified 3,621 of them with mock assertions. Our subsequent analysis will be based on these verified method invocations.

4 RQ1: How Frequently Do Developers Use Mock Assertions When Using Test Doubles?

Table 1 presents the demographics of our dataset. Among the 4,652 test cases that call at least one method of mock objects, 1,917 contain at least one mock assertion, representing 41% of the total. This observation underscores the widespread adoption of mock assertions by developers when using mock objects, emphasizing their significance in ensuring the correctness of software.

Notably, despite the prevalent usage of mock assertions and the frequent interaction between the SUTs and mock objects, we observed a relatively low verification rate for method calls. As indicated in the fifth column of Table 2, the verification ratio varies from 5% to 21% across our subjects, with Neo4j being the most prevalent and CXF being the least. On average, only 9% of method calls made to mock objects are verified by mock assertions. The result shows that verifying every method calls on test doubles with mock assertions is not the state of the practice. The disparity between

Table 2. Overview of Method Invocations on Test Doubles

Project	TDs Created	Unique Invocations[1]			Unique Methods[2]		
		Total	Stubbed	Verified	Total	Stubbed	Verified
Camel	1,159	2,966	1,982 (67%)	467 (16%)	470	274	126
CXF	1,328	4,323	3,191 (74%)	209 (5%)	511	344	74
Dubbo	404	1,934	1,266 (65%)	138 (7%)	313	161	66
Hadoop	3,221	16,612	9,289 (56%)	879 (6%)	1,268	599	194
Hazelcast	389	826	675 (82%)	74 (9%)	116	89	22
Kafka	1,188	5,513	3,196 (58%)	463 (8%)	537	309	134
MyBatis3	365	578	403 (70%)	76 (14%)	162	104	33
Neo4j	2,179	4,457	2,554 (57%)	917 (21%)	770	385	290
Ozone	482	1,241	929 (75%)	94 (8%)	151	102	40
Spring Boot	920	1,665	1,168 (70%)	221 (14%)	337	220	101
Storm	263	524	275 (52%)	83 (16%)	109	53	30
Total	11,898	40,639	24,928 (61%)	3,621 (9%)	4,628	2,576	1,080

1: Uniquely identified by test case, method signature, object ID, and stack trace.
2: Uniquely identified by method signature and mock object type.

the prevalent usage and the low verification ratio prompted us to conduct a deeper analysis to elucidate the specific usage patterns of mock assertions.

Interestingly, we found that only 1,032 (4%) of the stubbed invocations are verified, while 2,589 (17%) of the non-stubbed invocations are verified. A chi-square test [21] of independence was performed to examine the relation between the usage of stub and mock. The relation between these two variables was significant ($p < 0.05$). Within the same test double, developers are unlikely to verify the calls to the methods used for the role stub. While a test double can assume various roles, it is rare for these roles to be fulfilled by the same method.

RQ1 in Summary: Mock assertions are used in 41% of test cases using test doubles. However, only 9% (on average) of the method invocations on test doubles were verified by mock assertions. The result shows that verifying every method calls on test doubles with mock assertions is not the state of the practice. In addition, within the same test double, it is uncommon for a method to fulfill multiple roles.

5 RQ2: What Kinds of Method Calls Are Verified by Mock Assertions?

RQ2 aims to understand the types of methods developers verify using mock assertions. This section details our approach to identifying the characteristics of these methods and presents our findings.

5.1 Empirical Study Approach

As illustrated in Table 2, a total of 3,621 unique method invocations are verified by mock assertions, which correspond to 1,080 distinct methods. To gain deeper insights into the characteristics of verified methods (i.e., whose invocations are verified by mock assertions), we randomly sampled a statistically significant subset of 284 methods, ensuring a confidence level of 95% and a margin of error of 5%. To understand the difference between the verified methods and not verified methods, we further sampled 347 methods from the remaining 3,548 (4,628 - 1,080) for analysis. Such a sample size ensures a confidence level of 95% and a margin of error of 5%.

We employed an open coding approach [15] to systematically categorize these methods. Initially, we created tags based on JavaDoc, implementations, and comments associated with each selected method. These tags describe the roles and behaviors of the methods. Following this tagging

Table 3. Categorization of Methods Types Verified (or Not) by Mock Assertions. Chi-square tests of independence examined the relation between each category's membership and the method's verification.

Category[1]	Verified	Not Verified	$\chi^2 (p < 0.05)$	Description
External Resource	130 (46%)	30 (9%)	113.75	Interact with I/O, concurrency, database, etc.
State Mutator	79 (28%)	47 (14%)	19.91	Changes the internal fields of the object.
Callback	39 (14%)	2 (0.06%)	44.49	Event handlers, listeners, etc.
Accessor	41 (14%)	261 (75%)	231.10	Returns the internal state of the object.
Others	7 (3%)	10 (3%)	0.1036	Do not fall into any of the categories above.
Sample Size	284	347	Significant when $\chi^2 > 3.84$	CL=95%, ME=5%

* These categories may overlap. For example, a callback may also be a state mutator.

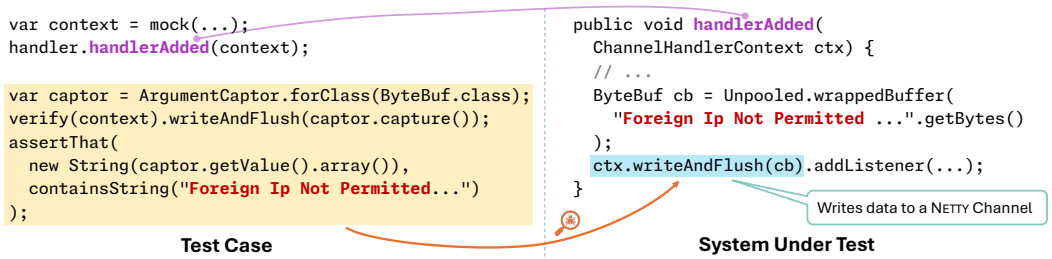


Fig. 2. A Mock Assertion Verifying an Invocation to a Method that Writes Data to a Netty Channel (Project Dubbo). Developers check the argument to ensure the correctness of the data being written.

process, we performed two rounds of axial coding. Specifically, we grouped method with similar tags together and came up with a higher-level tag to describe them. For example, we tagged `FileChannel.truncate()` with *File System* during open coding. Later in axial coding, it was merged into *I/O* and finally merged into *External Resource*. Two authors participated in the task and they discussed the results in their meetings. On disagreement, a third author joined to resolve the conflict. We finally got five categories from the coding process. These categories reflect the behaviors and roles of the methods whose invocations are verified (or not) by developers with mock assertions. After the coding process, we performed Chi-square tests of independence to examine whether the methods in each category are (un)likely to be verified.

5.2 Empirical Study Results

Table 3 shows the categorization of the method types we have identified. In total, we identified four categories of the methods. We present each of these categories in detail.

External Resources (130/284). This category encompasses methods that interact with external resources, which are crucial for enabling applications to communicate with the outside world. Specifically, it includes the methods that (1) manage file systems, transmit data over networks, generate logs, access streams, (2) interact with resources provided by operating systems (e.g., processes, threads, authentication, containers, and hardware), and (3) access external databases, manages database connections, executing SQL queries, and modifying data in key-value stores.

Developers use mock assertions to verify that methods interacting with external resources are called with the appropriate parameters such that anticipated interactions with the environment take place. For instance, Figure 2 illustrates an example from project Dubbo, where developers leverage mock assertions to check the correctness of data written to a Netty channel. In this example, the test

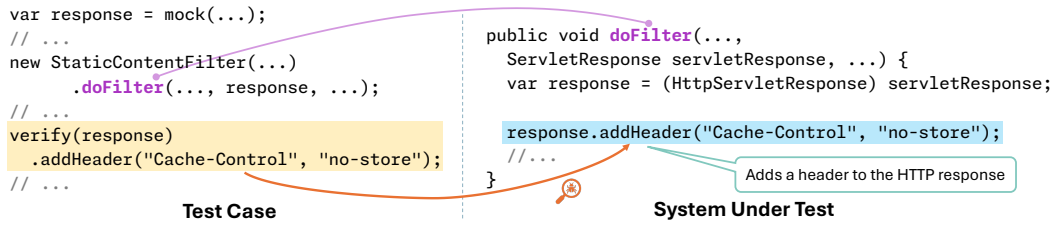


Fig. 3. A Mock Assertion Verifying an Invocation to a State Mutator that Inserts a Header to an HTTP Response (Project Neo4j). Developers check the correctness of the arguments.

case invokes the method `handlerAdded` in SUT with a mock `ChannelHandlerContext`. Inside the SUT, the error message is encoded into a buffer and subsequently written to the mock Netty channel by calling its method `writeAndFlush`. Back in the test case, developers first utilize a mock assertion to ensure that the method `writeAndFlush` is invoked. Then, they employ an `ArgumentCaptor` to extract the argument and verify that it contains the expected error message using a JUnit assertion. Such a mock assertion ensures the correct data has been transmitted over the network.

Among the 284 randomly sampled methods, 130 (46%) interact with external resources. In comparison, methods in this category only count for 9% of the methods that are not verified. The result of the Chi-square test was significant, which shows that methods interacting with external resources are more commonly verified. By checking the method calls to these methods with mock assertions, developers ensure the soundness of the integration points between the SUT and the external environment. Ultimately, this practice enhances test reliability and fosters confidence in the system's interactions with external dependencies.

State Mutators (79/284). This category includes methods that mutate the state of the test dependencies, which is essential for program state transitions within the system. Such methods include (1) setters that update the value of fields, and (2) domain-specific mutators that change the object's internal state.

Verifying method calls to state mutators is essential for ensuring that the correct state transitions have taken place. For example, Figure 3 shows an example from the Neo4j project. In the test case, the developer calls the method `doFilter` in SUT with a mock `response`. Inside the SUT, an HTTP header `"Cache-Control: no-store"` is added to the response via the method `addHeader`, which mutates a private field of `ServletResponse` by inserting the given header value. Afterwards, the developer uses a mock assertion to ensures that the method has been invoked with the expected key and value. Such a mock assertion ensure the correct header has been added.

Among the 284 randomly sampled methods, 79 (28%) are classified as mutator methods. In comparison, methods in this category only count for 14% of the methods that are not verified. The result of the Chi-square test was significant, which shows that state mutators are more commonly verified. By employing mock assertions in conjunction with these methods, developers can effectively test the correctness of state transitions within their applications. This practice reinforces the reliability of the system by ensuring that state changes do not lead to unintended side effects.

Callbacks (39/284). This category includes callback methods that function as event handlers or listeners, enabling applications to respond to events. Callback methods are invoked in response to user actions, system events, or the completion of asynchronous tasks. These methods may or may not interact with external resources or mutate the state of the test dependency, depending on the actual implementation.

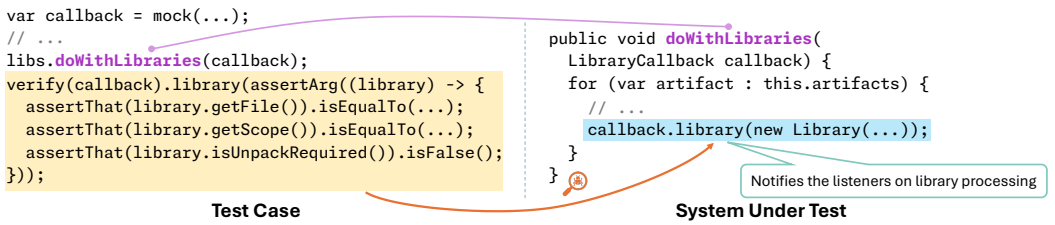


Fig. 4. A Mock Assertion Verifying an Invocation of a Callback Method Triggered During Library Processing (Project SpringBoot). Developers further check if the `library` passed to the callback is correct.

Developers verify invocations to callback methods to ensure that the event-handling pipeline is correctly established. Figure 4 illustrates an example from the SpringBoot project. In the test case, the developer invokes the method `doWithLibraries` in SUT with a mock `callback` to process library artifacts. Inside the SUT, the method `library` is called when each artifact has been processed to notify the components that are interested in library processing. In the test case, developers use a mock assertion to confirm that the callback method has been invoked. They also extract the argument and further validate it with three AssertJ assertions.

Among the 284 randomly sampled methods, 39 (14%) are classified as callback methods. In comparison, methods in this category only count for 0.06% of the methods that are not verified. The result of the Chi-square test was significant, which shows that callback methods are more commonly verified. Although this ratio may seem low, these methods are crucial for establishing the event-handling mechanism within applications. By validating the invocation of callback methods, developers can ensure that the SUT correctly fires events to interested components.

Accessors (41/284). This category includes methods that are used to retrieve an external state via the test dependency. Methods in this category include (1) methods that return a value based on the argument as a query key (e.g., `getHeader`), and (2) getters that solely return the value of a field.

Among the 284 randomly sampled methods, 41 (14%) fall under the accessor category. In comparison, methods in this category count for 75% of the methods that are not verified. The result of the Chi-square test was significant, which shows that accessor methods are not commonly verified. Although verifying method calls to accessors is not common, sometimes they can serve as a proxy for observing the actual execution path in the SUT and are important for revealing control flow discrepancies. Such cases will be discussed in Section 6 and Section 7.

RQ2 in Summary: Developers use mock assertions to verify methods in three categories: method interacting with external resources (46%), validating interactions with external systems; state mutators (28%), validating program state transitions; and callbacks (14%), ensuring event handlers are triggered properly. In comparison, accessors are unlikely to be verified.

6 RQ3: How Do the SUTs Interact With the Methods Verified by Mock Assertions?

In unit testing, the SUT frequently interacts with the test doubles [10] and developers use mock assertions to verify them [29, 39]. Therefore, RQ3 aims to identify the characteristics of the interactions between the SUT and the verified method calls.

6.1 Empirical Study Approach

To answer RQ3, we randomly sampled a subset of 348 method calls from the 3,621 unique method calls as shown in Table 2, ensuring a confidence level of 95% and a margin of error of 5%.

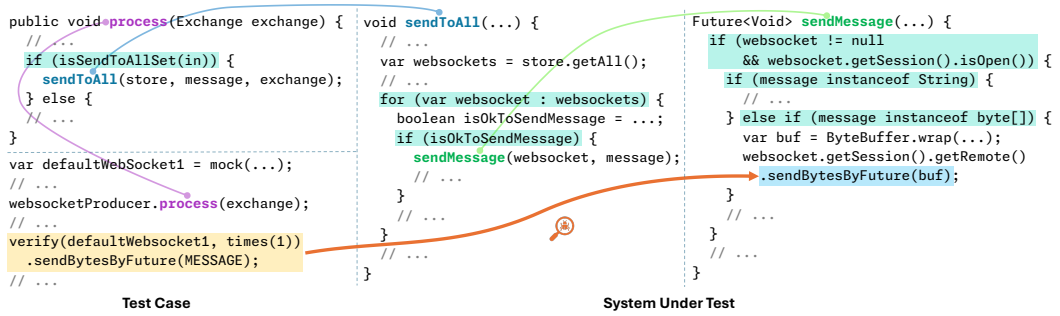


Fig. 5. A Mock Assertion Verifying a Method Call that is Made Conditionally (Project Camel). The method invocation is guarded by multiple branch conditions and loops.

Similarly as in RQ2, we adopted the open coding approach [15] to identify the common characteristics of the method invocations made to the test doubles. In RQ3, our focus shifts from the behavior of the methods themselves to the interactions between the SUT and the method invocations. To systematically document these interactions, we characterized each method call based on its specific role in the context of the SUT's execution. This involved analyzing the data flow and control flow patterns with each method call, allowing us to capture how information flow and control flow logic in the methods affect the method invocation within the SUT. Based on the observed behaviors and roles of the method calls, we applied descriptive tags to categorize them. Following this tagging process, we analyzed the tags to identify and merge similar categories, ultimately consolidating them into two major categories: one related to data flow and the other to control flow. These categories reflect the dual nature of the interactions, highlighting the significance of both the information transfer and the decision-making processes that influence the method calls.

6.2 Empirical Study Results

Among the 348 sampled method calls, we identified two primary categories of interactions between the SUT and the mocked method call. They are related to control flow and data flow, respectively.

Conditional Invocation (181/348). The first category of interactions we observe relates to the control flow characteristics of method calls. These calls are executed conditionally, depending on the test input or the internal states of the SUT. Examples of such conditional invocations include: (1) governed by `if`, `switch`, or loop constructs, (2) occurring within exception handlers, and (3) preceded by an early `return` or `throw`.

Figure 5 illustrates an example from the Camel project. In this test case, the developer calls the `process` method of the SUT. After executing method calls to `sendToAll` and `sendMessage`, the execution ultimately dispatches a call to `sendBytesByFuture` of the object `defaultWebSocket1`. This method call resides deep within nested branch conditions, governed by four `if` statements and a `for` loop. The developer employs a mock assertion to verify whether this method was invoked, ensuring the correctness of the `if` statements and the `for` loop. By using the presence of this method call as a proxy, developers can ascertain that the branch conditions (as highlighted in Figure 5) are correctly implemented, thereby enhancing the reliability of the SUT.

Among the 348 sampled method calls, 181 (52%) fall into this category. The frequencies of these conditional invocations serve as a proxy to the actual execution path. For example, a method call guarded an `if` statement is executed only when the conditions of the `if` statement is met. Verifying such method invocations with mock assertion can help ensure the correctness of branch conditions.

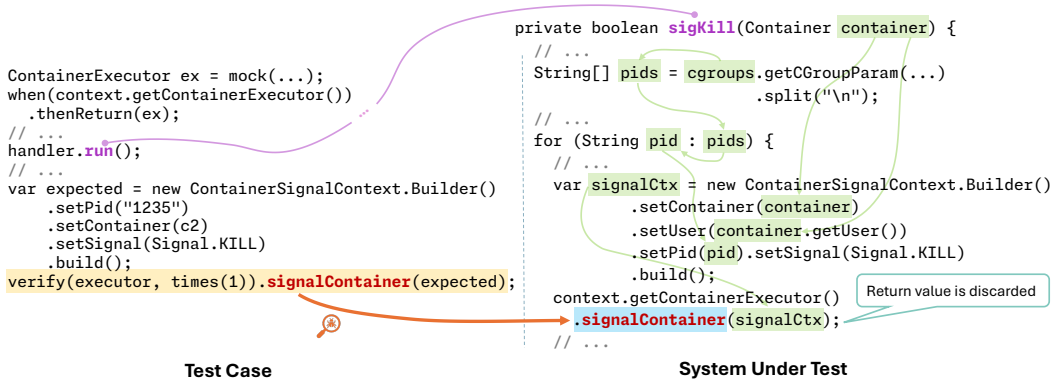


Fig. 6. A Mock Assertion Verifying a Method Call Acting as a Data Consumer (Project Hadoop). The method invocations uses data generated within the SUT while its return value is discarded.

As mentioned by a Hadoop developer in Pull Request #1146[2], *Method call verification is generally used for mocked methods so that we know the code path went through that.* In general, developers can validate the correctness of the implemented control flow logic by verifying the method calls that are made conditionally.

Data Consumers (179/348). The second category of interactions we observed pertains to the data flow aspect of method calls. Specifically, two characteristics are found for these method calls: (1) in the SUT, internal computation results are passed as arguments to these method calls, and (2) the method call does not return anything or the SUT discards the return value. Such method invocation consumes data generated by the SUT instead of producing new data for it. Therefore, we refer to such method calls as *data consumers*. These method invocations capture the internal computation results of the SUT, which are usually unobservable by test assertions. Therefore, developers verify such method invocations with mock assertions to observe such internal computation results.

Figure 6 illustrates an example in the Hadoop project. In the SUT, method `sigKill` constructs the argument `signalCtx` using results computed from the argument `container` and a private field `cgroups`. The data encapsulated in `signalCtx` is then passed to the method call `signalContainer`, which is responsible for terminating the execution of a container. Although the method returns a boolean value indicating whether the signal was successfully sent, this return value remains unused by the SUT. In this example, the method `signalContainer` is invoked in a fire-and-forget manner. Instead of producing new data for the SUT, the method call to `signalContainer` consumes data computed internally within the SUT to perform a side effect, namely, sending a signal to the container. In the corresponding test case, developers verify the method call to `signalContainer` using a mock assertion. Specifically, they check whether the argument passed to `signalContainer` matches the expected value to ensure that the SUT sends the correct signal to the container.

Among the 348 sampled method calls, 179 (51%) act as data consumers in the SUT. Generally, the SUT passes its internal states as arguments to method calls to perform side effects. Additionally, it often discards the returned data or simply checks the return value to confirm the success of the operation. During the verification of such method calls, developers typically capture the arguments to validate that the internal states passed to these calls are correct, ensuring that the intended side effects are achieved. Among the 179 method calls identified as data consumers, developers check

[2]Hadoop HDDS-1366: https://github.com/apache/hadoop/pull/1146#discussion_r312278997

that the arguments match the expected values in 149 (83%) cases. This phenomenon shows that verifying the correctness of arguments is a crucial aspect of testing data consumer methods.

RQ3 in Summary: Developers use mock assertions to verify method calls that are invoked conditionally (52%). The frequency of such method calls is a proxy of the correctness of the control flow logic in the SUT. Furthermore, developers verify method calls acting as data consumers (51%) to ensure the correctness of internal computation results.

7 RQ4: How Do Mock Assertions Complement Test Assertions in Fault Detection?

RQ4 aims to investigate the effectiveness of mock assertions in detecting potential faults. Specifically, we focus on the test cases where both mock assertions and test assertions are utilized. By examining the interplay between these two types of assertions, we aim to gain insights into how they collectively contribute to the fault detection capabilities of the test cases.

7.1 Empirical Study Approach

To achieve this goal, we employ mutation analysis [13] as a proxy to assess the adequacy of the tests in detecting potential faults. This technique creates mutants of the SUT by injecting artificial faults and checking if the test cases can "kill" them (i.e., the tests are expected to fail).

To understand how mock assertions complement test assertions in detecting potential faults, we performed mutation analysis on the variants of the test cases that use both types of assertions. Specifically, for each test case $\mathcal{T}$ that utilizes both types of assertions, we performed mutation analysis on the original tests and three of their variants:

- $\mathcal{T}$ (original): The original test case. It kills the mutants that either type of assertion can kill.
- $\mathcal{T}_{MA}$ (with mock assertions only): It kills the mutants that mock assertions in $\mathcal{T}$ can kill.
- $\mathcal{T}_{TA}$ (with test assertions only): It kills the mutants that test assertions in $\mathcal{T}$ can kill.
- $\mathcal{T}_{NA}$ (without any assertion): It kills the mutants that lead to a crash before any assertion in $\mathcal{T}$ is executed.

These variants exercise the same code in the SUT with the same test inputs, with the only difference in the assertions they use. In this case, we can isolate the contributions of each assertion type in fault detection. By definition, $Killed(\mathcal{T}) = Killed(\mathcal{T}_{MA}) \cup Killed(\mathcal{T}_{TA})$, and $Killed(\mathcal{T}_{NA}) \subseteq (Killed(\mathcal{T}_{MA}) \cap Killed(\mathcal{T}_{TA}))$.

To construct a dataset of such test cases, we adopted a semi-automatic approach. First, we executed the test cases to record their method invocations. Next, we selected the test cases that invoked both mock assertion APIs provided by Mockito and assertions APIs provided by assertions frameworks (e.g., JUnit). Out of the 4,652 test cases interacting with test doubles (Table 1), we identified 1,071 such test cases. However, it is important to note that invoking both types of assertion APIs does not necessarily mean the test case uses both types of assertions. This is because some JUnit assertions are dependent on mock assertions. As illustrated in Figure 1, a JUnit assertion can predicate a value captured by mock assertions. In such cases, these JUnit assert statements are considered part of mock assertions. Distinguishing such assert statements requires understanding the role of the assert statements. Therefore, we opted for a manual approach. Specifically, we randomly sampled 283 test cases out of the 1,071 for manual review. Such a sample size ensures a confidence level of 95% and a margin of error of 5%. During the manual review, we found that in 43 test cases, all the JUnit assertions were dependent on mock assertions. In other words, there were no test assertions in these test cases. Consequently, they were excluded from the mutation analysis. For the remaining 240 test cases, we removed the test assertions to create $\mathcal{T}_{MA}$, removed the mock assertions to create $\mathcal{T}_{TA}$, and removed all the assertions to create $\mathcal{T}_{NA}$. Notably, we kept the

Table 4. Distribution of Mutants Types Covered by Selected Test Cases

Mutation Operatior	# Mutants	Mutation Operator	# Mutants	Mutation Operator	# Mutants
NonVoidMethodCall	2,085	ArgumentPropagation	349	BooleanFalseReturnVals	46
RemoveConditional	1,931	NakedReceiver	203	PrimitiveReturns	45
NegateConditionals	959	NullReturnVals	199	RemoveSwitch	38
MemberVariable	844	ConditionalsBoundary	95	Increments	8
InlineConstant	598	Math	88	RemoveIncrements	8
VoidMethodCall	552	EmptyObjectReturnVals	83	Switch	7
ConstructorCall	357	BooleanTrueReturnVals	83	InvertNegs	1
↪	↪	↪	↪	**Total**	8,579

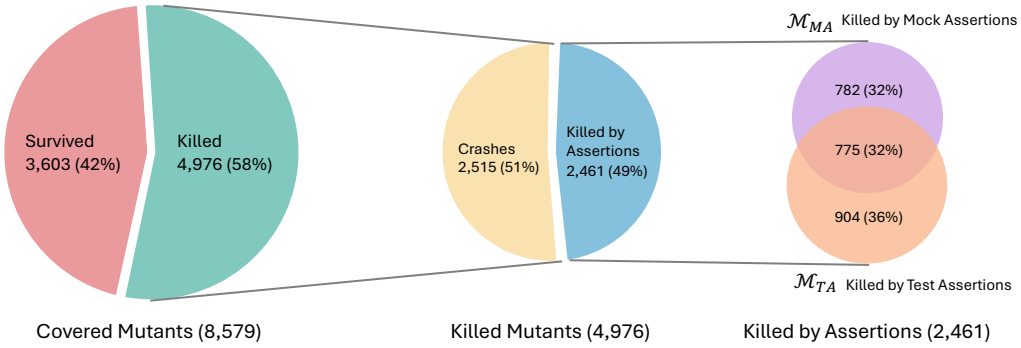


Fig. 7. Breakdown of the Covered Mutants by Each of the Assertion Types

inlined method calls when removing test assertions. For example, `assertEquals(..., sut.foo())` was turned into `sut.foo()` to preserve the coverage of the test case.

Subsequently, we ran the PIT mutation testing tool [7] on the variants of these test cases. We enabled all the builtin mutation operators of PIT to increase the diversity of the injected faults. For the killed mutants, we attribute them to the two assertion types as follows:

- $\mathcal{M}_{MA} = Killed(\mathcal{T}_{MA}) - Killed(\mathcal{T}_{NA})$ is the set of mutants killed by mock assertions.
- $\mathcal{M}_{TA} = Killed(\mathcal{T}_{TA}) - Killed(\mathcal{T}_{NA})$ is the set of mutants killed by test assertions.

During the mutation analysis, PIT seeded 26,481 faults into the SUTs, among which 8,579 were covered by the test cases. Since all these variants share the same test inputs and execution paths, they cover the same set of mutants. Table 4 show the distribution of the 8,579 covered mutants. They are produced by 21 mutation operators, with `NonVoidMethodCall` being the most frequent and `InvertNegs` being the least. Then, our analysis will focus on these covered mutants to evaluate the effectiveness of different assertion types in detecting injected faults.

7.2 Empirical Study Results

Figure 7 shows the distribution of the 8,579 mutants covered by the 240 test cases. There are 4,976 mutants killed by at least one of the variants, accounting for 58% of the covered mutants. Among the 4,976 killed mutants, 2,515 (51%) are killed by $\mathcal{T}_{NA}$. These mutants lead to crashes when the SUT is running, they can be detected even if no assertion was written. The remaining 2,461 (49%) mutants are not killed by $\mathcal{T}_{NA}$ but killed by any of $\mathcal{T}_{MA}$ and $\mathcal{T}_{TA}$. We find that mock assertions and test assertions are complementary in detecting potential faults. As shown in Figure 7, 68%

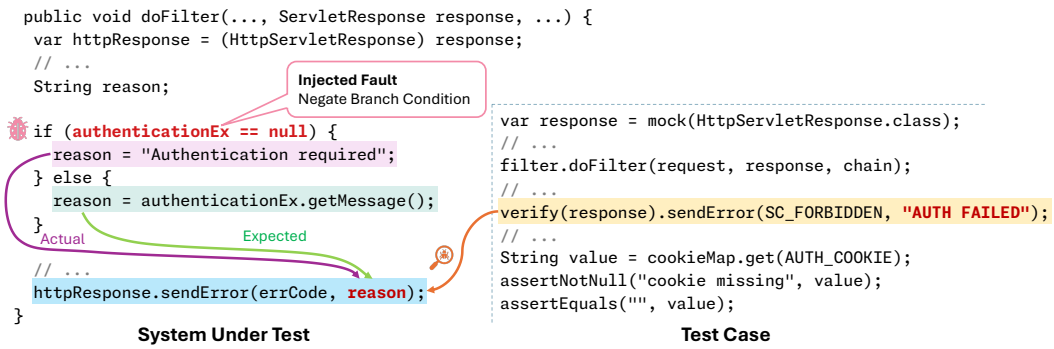


Fig. 8. An Incorrect Branch Condition Implementation Detected by a Mock Assertion (Project Hadoop). The method `sendError` interacts with external resources. The method invocation is a data consumer in the SUT.

Table 5. Distribution of the Mutants that are Detected by Mock Assertions Only

Method Behavior[1]	**# Mutants**	**Interactions with SUT**[2]	**# Mutants**
External Resource	114 (44%)	Conditional Invocation	198 (77%)
State Mutator	99 (38%)		
Callback	41 (16%)	Data Consumer	115 (45%)
Accessor	12 (5%)		
Sample Size	258 (CL=95%, ME=5%)		258 (CL=95%, ME=5%)

1 The overlaps between External Resource and State Mutators is 1, and the overlap between State Mutator and Callback is 7.
2 The overlap between these two categories is 83.

of the mutants are killed by only one assertion type. Notably, 782 of 1,557 (50%) of the injected faults detected by mock assertions were not detected by test assertions, and the Jaccard similarity between $\mathcal{M}_{MA}$ and $\mathcal{M}_{TA}$ is only 32%. The result shows that mock assertions can complement test assertions in detecting potential faults.

The complementariness is also found at test case level. In 109 (45%) of the 240 test cases, mock assertions and test assertions killed at least one mutant that the other assertion type did not kill. Moreover, in 138 (58%) of the 240 test cases, mock assertions killed at least one mutant that test assertions did not kill. Mock assertions and test assertions complement each other in these test cases. This shows the importance of mock assertions in assuring software quality.

Figure 8 shows an example in project Hadoop where mock assertions detect a fault that test assertions do not detect. In the SUT, depending on whether `authenticationEx` is `null`, different strings containing the `reason` for authentication failure are sent to the HTTP client via the method `sendError`. This method interacts with *external resources* and it is a candidate to be verified by mock assertions. In addition, such a method invocation is a *data consumer* that captures an internal computation result. During mutation analysis, an injected fault negates the condition of the `if` statement, which makes the SUT send a different `reason` to the client via `sendError`. Since the `reason` is sent over the internet and is never returned to the test case, test assertions cannot observe such a difference. However, mock assertions enable developers to check the arguments passed to the method `sendError`. Specifically, developers expect the second argument to be `"AUTH FAILED"`. During test execution, the mock assertion failed since the second argument for this method call was `"Authentication required"`. In this example, mock assertions complement test assertions by ensuring the correctness of important side effects and internal computation results.

To get a better understanding of the 782 injected faults that were only detected by mock assertions, we randomly sampled 258 of them for manual analysis, such a sample size ensures a confidence level of 95% and a margin of error of 5%. Specifically, we followed the criteria in RQ2 and RQ3 to categorize the verified method calls of the failing mock assertions. Table 5 shows the distribution of the 258 mutants. Among these sampled mutants, 246 (95%, 114+99+41-1-7) were detected by a mock assertion verifying a method in the category of *external resources*, *state mutator*, and *callback*. These methods produce important side effects on the test dependencies and external environments. The incorrect number of these method calls or incorrect arguments passed to these method calls can cause undesired side effects. Mock assertions help developers ensure these side effects are produced correctly. In addition, 230 (89%, 198+115-83) mutants were detected by a mock assertion verifying a method call that is one of or both *conditional invocation* and *data consumer*. Mock assertions verifying these method invocations ensure the correctness of the control flow logic and internal computation results. Notably, there are 12 mutants detected by mock assertions that verify an *accessor*, which seems to misalign with our findings in RQ2. However, we found 10 of them fall into the category of *conditional invocation*. Although accessors do not produce side effects, the presence or frequencies of their invocations can help developers validate the control flow logic in the SUT.

RQ4 in Summary: We observed a minor overlap (32%) between the faults detected by mock assertions and test assertions. Half of the faults detected by mock assertions were not detected by test assertions. Mock assertions complement test assertions by ensuring the desired side effects have been produced, validating control flow logic, and checking internal computation results.

8 Discussion

In this section, we summarize our major findings and provide advice for future researchers and developers. In addition, we discuss the threats affecting the validity of our findings.

8.1 Implications

Leverage Mock Assertions for Stronger Test Oracles. Our study demonstrates that mock assertions complement test assertions by ensuring the desired side effects have been produced, validating control flow logic, and checking internal computation results. Developers should utilize mock assertions to validate program behaviors that are unobservable by traditional test assertions. Furthermore, while automated test generation techniques often aim for higher test coverage through the use of test doubles [3–5, 30, 33], future research should explore the generation of mock assertions to create stronger test oracles in the generated tests.

Avoid Aggressive Mocking. Despite some automated test generation techniques generating mock assertions [1, 31], they adopt a brute-force approach that results in an excessive number of assertions, leading to fragile tests [27]. Indeed, our findings indicate that such aggressive use of mock assertions does not align with typical developer practices. Based on our findings, we recommend three strategies for future researchers and developers to pinpoint critical method invocations during test execution, and generate mock assertions to verify them.

(1) **Prioritize methods interacting with external resources, methods mutating program states, and callbacks.** In RQ2, we identified three categories of methods that developers commonly verify with mock assertions. (a) methods interacting with external resources, ensure correct interactions with the external environment; (b) methods mutating program states, ensure correct program state transitions; and (c) callbacks, ensure event handlers are triggered properly. In general, developers may consider verifying such method invocations

to ensures the desired side effects have been produced. Future research may explore automatically identifying such method calls, providing candidates for verification through mock assertions.

(2) **Focus on method invocations that distinguish execution paths.** In RQ3, we found that developers often verify method invocations executed conditionally based on test inputs or SUT states. The presence/absence/frequencies of such method invocations can serve as a proxy for observing the actual execution path. By enforcing the desired execution paths, verifying such method invocations can reveal potential control flow discrepancies. Developers and researchers working on test generation should consider verifying method calls executed in specific branches, especially when the branches are difficult to reach. Additionally, Zhu et al. [38] noted that such mock assertions can enhance generated stubs.

(3) **Capture and validate internal computation results.** In RQ3, we found that developers often verify method calls acting as data consumers within the SUT, which helps validate the internal computation results. For test generation techniques, beyond identifying these method calls, it is also important to generate assertions that predicate the arguments for such invocations. Developers are encouraged to capture the key internal computation results for validation.

These strategies can even be used in combination to generate more effective oracles with fewer mock assertions. For example, a developer can kill two birds with one stone by verifying a method call living in deep nested branches that also interacts with external resources. Future research can explore mechanisms (e.g., machine learning) to holistically consider these strategies to generate concise yet effective mock assertions.

Relation between Purity and Mock Assertions. In RQ2, we identified categories of methods that either have their behavior influenced by external states or produce side effects. These methods are often considered impure. Additionally, in RQ4, we discovered that mock assertions are particularly effective in detecting faults related to side effects. These findings indicate that developers frequently use mock assertions to verify the behavior of impure methods. Future research should explore the relationship between method purity and the use of mock assertions.

8.2 Threats to Validity

External Validity. Our empirical findings might not generalize to other Java projects. To mitigate this threat, we selected large-scale, popular Java projects that span various application domains. The quality and diversity of our subjects bolsters our findings generalizing to a wider range of scanarios. Also, We selected subjects using Mockito to create and configure test doubles in this paper. There are also other frameworks such as EasyMock [9], PowerMock[3] [23], and jMock [14]. Developers can have slightly different practices when using these frameworks due to the difference between their functionalities. Such differences were not covered by our study. Indeed, as reported by recent studies [10, 29, 35], Mockito is the most frequently adopted framework, and it is used in over 80% of the projects that uses a mocking framework. As a result, our findings can be applied to most of the scanrios where mock assertions are used.

In addition, in RQ4, we considered only the test cases where both types of assertions are used. This is because developer tend to use both assertion types to complement each other [22]. However, there may be cases where the two assertion types complement each other in different test cases. They exercise different code paths in the SUT with different test inputs, which poses challenges in correctly attributing a killed mutant to the assertions (rather than the test inputs). Therefore, we

[3]PowerMock can use Mockito as its backend.

did not include such test case for analysis, and our results do not reflect the complementariness of the two assertion types in such test cases. It is an important future work to investigate how mock assertions and test assertions complement each other in different test cases.

Internal Validity. As is the case with most empirical studies, there is a potential for errors introduced by manual analysis. Since the characterization of mock assertions relies on human judgment, there is a risk of misinterpreting or mischaracterizing certain mock assertions. These errors could lead to inaccurate characterizations of mock assertions. To mitigate this threat, the authors cross-checked the results produced by manual analysis and discussed their findings during meetings. Additionally, we have released our experimental data for readers to validate the results.

During data collection we inferred the SUT following the pattern used by the JUnit test runner and related work [32, 39]. However, this strategy might not accurately identify the SUT when developers do not follow the naming convention. To mitigate this threat, we manually reviewed a subset of the the test cases in our subjects where this naming convention was not followed. We found only 5% of the cases were false negatives, the impact of this threat is minor for our study.

Construct Validity. We used mutation analysis as a proxy to assess the fault detection capabilities of two assertion types in RQ4. However, the mutants generated by PIT [7] might not accurately represent the types of bugs encountered in real-world software development. To mitigate this threat, we enabled all the mutation operators in PIT to maximize the diversity of injected faults. Nevertheless, evaluating the mock assertions using real-world bugs is an important future work.

9 Related Work

Test doubles enable developers to simulate test dependencies in controlled ways, making it an important technique to support unit testing. In recent years, several studies (discussed below) have explored the usage of test doubles and developed techniques to facilitate their use. In this section, we discuss the most relevant studies that focus on the applications of test doubles.

Empirical Studies in Test Doubles Several studies have investigated the usage of test doubles in software testing. Marri et al. [18] examined the benefits of test doubles in testing file-system-dependent software, highlighting their ability to ease the unit testing process. They emphasized the need for automated identification of APIs requiring mocking. Zhu et al. [39] addressed this need by identifying code-level characteristics for mocking decisions and developing MockSniffer, a machine learning-based technique for recommending mocking decisions to developers. Mostafa and Wang [19] analyzed the usage of mocking frameworks in a vast number of open-source Java projects, revealing that while mock objects are widely used, only a subset of test dependencies are mocked. Spadini et al. [28] explored developers' mocking decisions and found that classes contributing to testing difficulties are often mocked. They further investigated the evolution of mocking framework usage [29], highlighting the frequent evolution of API usage related to mock assertions. Fazzini et al. [10] specifically studied the usage of test doubles in Android testing and identified the potential issues they introduce.

These empirical studies offer insights into developers' practices in using test doubles. They provide statistical evidence on the significance of test doubles and identify challenges that motivate further exploration. On the same theme, our study delves into the usage of mock assertions, aiming to provide guidance for developers and support future research with empirical evidence.

Test Double Automation Techniques The research community has made efforts to automate the creation and maintenance of test doubles. Saff et al. [24, 25] proposed a technique that generates test doubles by capturing interactions between the SUT and its dependencies during system tests. Tiwari et al. [31] developed Rick, which monitors SUT execution in a production environment

and generates mock-based test cases to validate common production usage. Wang et al. [34] introduced an automated refactoring technique that migrates inheritance-based mock objects to mocking frameworks. Other studies have focused on enhancing specific parts of tests involving mocking. AutoMock, proposed by Alshahwan et al. [2], employs symbolic execution to infer post-conditions that must be met by the return values of stubs. Fazzini et al. developed MOKA [11], which collects and generates reusable mock objects for testing Android applications. Zhu et al. proposed StubCoder [38] to generate and repair stub code for regression testing purposes. Similarly, Tung et al. proposed AUTS [33], which leverages stub code to generate tests for achieving higher code coverage for C/C++ programs. On the same theme, Li et al. proposed ARUS [16] to automatically remove unnecessary stubs from test suites. Additionally, domain-specific mock object generation techniques have been proposed for file systems [3, 18], databases [30], and networking [4, 6, 26, 36].

These techniques aim to improve test efficiency, increase coverage, and enhance maintainability by generating tests with test doubles. However, none of these techniques effectively identified the method invocations that should be verified by mock assertions. Our study aims to bridge this gap by offering insights and empirical evidence to guide future research in developing such a technique.

10 Conclusion and Future Work

In this paper, we conducted the first empirical study to understand the usage of mock assertions in practice. By analyzing the usage of mock assertions in 4,652 test cases in 11 open-source Java projects, we found that although mock assertions are used by 41% of the test cases that use test doubles, verifying all the method calls to test doubles with mock assertions is not the state of the practice. In contrast, developers only verify a small part of method calls. For the verified method invocations, we identified three categories of methods whose invocations are commonly verified by developers and two types of common interactions between the SUT and the verified method invocations. Last but not least, we found mock assertions complement test assertions by ensuring the desired side effects have been produced, validating control flow logic, and checking internal computation results. We hope our findings can provide guidance and empirical evidence for future research in this area. Developers can also benefit from our findings and make better use of mock assertions when writing tests.

An important future work following this study is to explore automated mechanisms to identify the method invocations that should be verified by mock assertions. One of the key challenges is to balance between the strength (i.e., low false negative) and robustness (i.e., low false positive) of the test oracle. Such a technique can reduce the burden on developers when creating test oracles. In addition, integrating such identification mechanisms with automated test generation techniques enables them to generate proper mock assertions to strengthen the tests.

11 Data Availability

Our experimental data is available at https://doi.org/10.5281/zenodo.14695509 [37]. The dataset is licensed under the Creative Commons Attribution 4.0 International License.

Acknowledgments

We would like to express our appreciation to the anonymous reviewers for their insightful and constructive comments. We would also like to thank the developers of our studied projects for their generous contributions, which made this research possible. This work is supported by Hong Kong Research Grants Council / General Research Fund (HKSAR RGC/GRF, grant no. 16205722), the FRQNT/NSERC NOVA program (grant no. 2024-NOVA-346499), and National Natural Science Foundation of China (grant no. 62372219).

References

[1] Agitar Technologies. [n. d.]. *Agitar One Test Generator.* Retrieved August 2024 from http://www.agitar.com/solutions/products/agitarone.html

[2] Nadia Alshahwan, Yue Jia, Kiran Lakhotia, Gordon Fraser, David Shuler, and Paolo Tonella. 2010. AUTOMOCK: Automated Synthesis of a Mock Environment for Test Case Generation. In *Practical Software Testing: Tool Automation and Human Factors, 14.03. - 19.03.2010 (Dagstuhl Seminar Proceedings, Vol. 10111)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2010/2618/

[3] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM, 79–90. doi:10.1145/2642937.2642986

[4] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 155–165. doi:10.1145/2786805.2786828

[5] Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private API Access and Functional Mocking in Automated Unit Test Generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*. IEEE Computer Society, 126–137. doi:10.1109/ICST.2017.19

[6] Thilini Bhagya, Jens Dietrich, and Hans W. Guesgen. 2019. Generating Mock Skeletons for Lightweight Web-Service Testing. In *26th Asia-Pacific Software Engineering Conference, APSEC 2019*. IEEE, 181–188. doi:10.1109/APSEC48747.2019.00033

[7] Henry Coles. [n. d.]. *PIT Mutation Testing.* Retrieved August 2024 from https://pitest.org

[8] Mockito contributors. [n. d.]. *Mockito framework site.* Retrieved August 2024 from https://mockito.org

[9] EasyMock Contributors. [n. d.]. *EasyMock.* Retrieved August 2024 from https://easymock.org

[10] Mattia Fazzini, Chase Choi, Juan Manuel Copia, Gabriel Lee, Yoshiki Kakehi, Alessandra Gorla, and Alessandro Orso. 2022. Use of Test Doubles in Android Testing: An In-Depth Investigation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2266–2278. doi:10.1145/3510003.3510175

[11] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. 2020. A Framework for Automated Test Mocking of Mobile Apps. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 1204–1208. doi:10.1145/3324884.3418927

[12] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *Proceedings of the 11th International Conference on Quality Software, QSIC 2011, Madrid, Spain, July 13-14, 2011*, Manuel Núñez, Robert M. Hierons, and Mercedes G. Merayo (Eds.). IEEE Computer Society, 31–40. doi:10.1109/QSIC.2011.19

[13] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[14] jMock Contributors. [n. d.]. *jMock.* Retrieved August 2024 from https://jmock.org

[15] Sarah Lewis. 2015. Qualitative inquiry and research design: Choosing among five approaches. *Health promotion practice* 16, 4 (2015), 473–475.

[16] Mengzhen Li and Mattia Fazzini. 2024. Automatically Removing Unnecessary Stubbings from Test Suites. arXiv:2407.20924 [cs.SE] https://arxiv.org/abs/2407.20924

[17] Tim Mackinnon, Steve Freeman, and Philip Craig. 2001. *Endo-Testing: Unit Testing with Mock Objects.* Addison-Wesley Longman Publishing Co., Inc., USA, 287–301.

[18] Madhuri R. Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. In *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009*. IEEE Computer Society, 149–153. doi:10.1109/IWAST.2009.5069054

[19] Shaikh Mostafa and Xiaoyin Wang. 2014. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *2014 14th International Conference on Quality Software*. IEEE, 127–132. doi:10.1109/QSIC.2014.19

[20] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 815–816. doi:10.1145/1297846.1297902

[21] Karl Pearson. 1992. *On the Criterion that a Given System of Deviations from the Probable in the Case of a Correlated System of Variables is Such that it Can be Reasonably Supposed to have Arisen from Random Sampling.* Springer New York, New York, NY, 11–28. doi:10.1007/978-1-4612-4380-9_2

[22] Anthony Peruma, Mohamed Wiem Mkaouer, Khalid Almalki, Christian D. Newman, Ali Ouni, and Fabio Palomba. 2024. *Test Smells.* Retrieved August 2024 from https://testsmells.org/pages/testsmells.html

[23] Power Contributors. [n. d.]. *PowerMock.* Retrieved August 2024 from https://powermock.github.io

[24] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. 2005. Automatic test factoring for java. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*. ACM, 114–123. doi:10.1145/1101908.1101927

[25] David Saff and Michael D. Ernst. 2004. Mock object creation for test factoring. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04*. ACM, 49–51. doi:10.1145/996821.996838

[26] Susruthan Seran, Man Zhang, and Andrea Arcuri. 2023. Search-Based Mock Generation of External Web Service Interactions. In *Search-Based Software Engineering - 15th International Symposium, SSBSE 2023, San Francisco, CA, USA, December 8, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14415)*, Paolo Arcaini, Tao Yue, and Erik M. Fredericks (Eds.). Springer, 52–66. doi:10.1007/978-3-031-48796-5_4

[27] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 201–211. doi:10.1109/ASE.2015.86

[28] Davide Spadini, Mauricio Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock?: an empirical study on mocking practices. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 402–412. doi:10.1109/MSR.2017.61

[29] Davide Spadini, Mauricio Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing java systems - Why and how developers use them, and how they evolve. *Empir. Softw. Eng.* 24, 3 (2019), 1461–1498. doi:10.1007/s10664-018-9663-0

[30] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: automated test generation for database applications via mock objects. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 289–292. doi:10.1145/1858996.1859053

[31] Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2022. Mimicking Production Behavior with Generated Mocks. *CoRR* abs/2208.01321 (2022). doi:10.48550/ARXIV.2208.01321 arXiv:2208.01321

[32] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE]

[33] Lam Nguyen Tung, Nguyen Vu Binh Duong, Khoi Nguyen Le, and Pham Ngoc Hung. 2024. Automated test data generation and stubbing method for C/C++ embedded projects. *Autom. Softw. Eng.* 31, 2 (2024), 52. doi:10.1007/S10515-024-00449-6

[34] Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, and Sunny Wong. 2021. An automatic refactoring framework for replacing test-production inheritance by mocking mechanism. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 540–552. doi:10.1145/3468264.3468590

[35] Lu Xiao, Gengwu Zhao, Xiao Wang, Keye Li, Erick Lim, Chenhao Wei, Tingting Yu, and Xiaoyin Wang. 2024. An empirical study on the usage of mocking frameworks in Apache software foundation. *Empir. Softw. Eng.* 29, 2 (2024), 39. doi:10.1007/S10664-023-10410-Y

[36] Linghao Zhang, Xiaoxing Ma, Jian Lu, Tao Xie, Nikolai Tillmann, and Peli de Halleux. 2012. Environmental Modeling for Automated Cloud Application Testing. *IEEE Softw.* 29, 2 (2012), 30–35. doi:10.1109/MS.2011.158

[37] Hengcheng Zhu, Valerio Terragni, Lili Wei, Sheng-Chi Chung, Jiarong Wu, and Yepang Liu. 2025. *Experimental Data: Understanding and Characterizing Mock Assertions in Unit Tests*. doi:10.5281/zenodo.14695510

[38] Hengcheng Zhu, Lili Wei, Valerio Terragni, Yepang Liu, Shing-Chi Cheung, Jiarong Wu, Qin Sheng, Bing Zhang, and Lihong Song. 2024. StubCoder: Automated Generation and Repair of Stub Code for Mock Objects. *ACM Trans. Softw. Eng. Methodol.* 33, 1 (2024), 16:1–16:31. doi:10.1145/3617171

[39] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 436–447. doi:10.1145/3324884.3416539

Received 2024-09-13; accepted 2025-01-14