From Inheritance to Mockito: An Automatic Refactoring Approach

Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, Sunny Wong

***Abstract*—Unit testing focuses on verifying the functions of individual units of a software system. It is challenging due to the high inter dependencies among software units. Developers address this by mocking—replacing the dependency by a "fake" object. Despite the existence of powerful, dedicated mocking frameworks, developers often turn to a "hand-rolled" approach—inheritance. That is, they create a subclass of the dependent class and mock its behavior through method overriding. However, this requires tedious implementation and compromises the design quality of unit tests. This work contributes a fully automated refactoring framework to identify and replace the usage of inheritance by using Mockito—a well received mocking framework. Our approach is built upon the empirical experience from *five* open source projects that use inheritance for mocking. We evaluate our approach on *nine* other projects. Results show that our framework is efficient, generally applicable to new datasets, mostly preserves test case behaviors in detecting defects (in the form of mutants), and decouples test code from production code. The qualitative evaluation by experienced developers suggests that the auto-refactoring solutions generated by our framework improve the quality of the unit test cases in various aspects, such as making test conditions more explicit, as well as improved cohesion, readability, understandability, and maintainability with test cases. Finally, we submit 23 pull requests containing our refactoring solutions to the open source projects. It turns our that, 9 requests are accepted/merged, 6 requests are rejected, the remaining requests are pending (5 requests), with unexpected exceptions (2 requests), or undecided (1 request). In particular, among the 21 open source developers that are involved in the reviewing process, 81% give positive votes. This indicates that our refactoring solutions are quite well received by the open source projects and developers.**

***Index Terms*—software refactoring, software testing, mocking**

I. INTRODUCTION

Unit testing is an important phase of testing that focuses on individual units of a software system [1]. However, the challenge to unit testing is that software elements are inter-dependent on each other [1], [2]. That is, when testing one function, we have to consider its dependencies to other functions. This hinders our ability to test easily and promptly. For example, the function under test (*FUT*) may depend on an external database that has not been deployed. This challenge also applies to debugging — if a unit test fails, it is unclear whether the failure is caused by the fault in *FUT* or its dependent functions.

A general methodology to address this challenge is isolating the core *FUT* from its dependencies through mocking [3], [4], i.e., replacing the dependency by a "fake" object. For example, instead of waiting until the external database is deployed, developers create a "fake" database with dummy data populated in a local file system and control its behavior to serve for the testing purposes. There are various dedicated mocking frameworks, such as *easyMock*, *Mockito*, and *PowerMock* [5]–[7], which provide well constructed solutions to isolate *FUT* from its dependencies. Specifically, they provide powerful functions allowing developers to easily create mock objects, control their behavior, and verify the execution/status of the mock objects. These frameworks work together with classic automated unit testing frameworks, such as JUnit [8] and PyUnit [9].

Despite the existence of powerful mocking frameworks, developers often turn to a "hand-rolled" approach—inheritance [10]. That is, to create a "fake" object, developers create a subclass of the dependent production class and control its behavior through method overriding. For example, in the *fourteen* open source projects examined in this study (Section III-A and Section V-B), developers already adopt an existing mocking framework for testing their systems. However, in about half of the cases when mocking is potentially needed, developers still use inheritance instead of using a mocking framework. The problem is that inheritance is not intended for mocking. As such, it requires tedious implementation when being used for this purpose. In addition, it may compromise the design quality of unit tests and lead to maintenance difficulties in the test cases [11]–[14]. More specifically, as illustrated in Section II, inheritance has the following drawbacks compared to using a mocking framework such as Mockito. First, inheritance implicit test condition and blurred test logic. Second, inheritance make test code couples with the production code and difficult to maintain. Third, inheritance separates the mocking behavior from the test case that leverages it and makes test design incohesive.

The goal of this work is to develop a fully automated refactoring framework to identify and replace the usage of inheritance by using Mockito for mocking in unit testing. We choose Mockito because it is one of the most well received mocking framework for Java projects [14]. It is adopted in both commercial and open source projects [3]. The key challenge is to preserve the test behaviors before and after the refactoring. To overcome this challenge, we first conduct an empirical study (Section III) involving *five* real-life, open-source projects as the learning dataset. The goal is to gain empirical experience of whether it is feasible and how to perform refactoring following an automated procedure. Based on the empirical observations, we formalize the problem definition of auto-refactoring to replace inheritance by using Mockito (Section III-C). Next, we propose a fully automated refactoring framework and

implement it as an Eclipse-Plugin (Section IV). This framework first identifies all feasible refactoring candidates and then performs the refactoring on each candidate for a given project.

We perform both quantitative and qualitative evaluation (Section V) of the proposed framework using another *nine* open-source projects.

The quantitative evaluation shows that Our approach derived from the manual experience from five open source projects is highly applicable to fourteen new open source projects. Second, Our refactoring solutions especially reduce the coupling between the test and production code. Third, The test cases show high-level of behavior preservation before and after the refactoring. Fourth, it is highly efficient to apply our automatic refactoring tool on real-life projects.

In addition, the qualitative evaluation contains two separate studies. In the first study, we invite experienced, full-time developers from a company to evaluate the quality and value of our refactoring solutions. In particular, we compare our auto-generated refactoring solutions with manual refactoring solutions created by the participating developers. The study results show that the auto-refactoring solutions generated by our approach are of good design quality and provide various benefits for improving test code design.

In the second qualitative study, we submit a total of 23 pull requests containing our auto-generated refactoring solutions to the fourteen projects involved in this study. The goal is to evaluate how well the open source projects actually receive the refactoring solutions generated by our approach. We evaluate this through the response rate and turn-around time to our pull requests, especially when comparing to the average response rate and turn-around time in the projects. In addition, we also examine the acceptance rate based on pull requests and based on developers that provided comments. The results show that our refactoring solutions are quite well received from the above metrics in open source projects. Finally, we summarize the comments from the developers. These comments underscore the strength of our tool, reveal concerns of developers who rejected our pull requests, as well as highlight future improvement opportunities identified in our approach.

In summary, this work makes the following contributions:

- An empirical study involving *five* open-source projects investigating whether it is feasible and how to automatically replace inheritance by Mockito for mocking.
- A fully automated refactoring framework and its Eclipse-Plugin implementation to identify feasible refactoring candidates and perform the refactoring on each candidate.
- Quantitative and qualitative evaluation of the proposed framework on *nine* open-source projects.

This work is an extension of our prior work [15]. The extension is in four key aspects:

1) We improved our approach by loosening two refactoring pre-conditions, which involve new public methods and generic types in test subclasses for mocking. These cases are considered infeasible for refactoring in the conference version [15]; in this journal version, our approach can handle cases with these two pre-conditions. The feasibility of our approach has increased by 3% (from 40% to 43% on the testing dataset) based on our experiment.
2) In [15], we only evaluated our refactoring approach on 4 projects. In this work, we have significantly extended our evaluation for general applicability. In particular, to test the general feasibility, we applied our refactoring pre-conditions on 182 Apache java projects, and found that 26%—a total of 2,609—test sub-classes qualify as feasible refactoring candidates for our approach. In addition, we more than doubled the number of projects that actually apply our auto-refactoring approach. The results show that our approach has 82% success rate on feasible cases.
3) In this study, we added a new research question, namely *RQ7: How well do the open source projects receive the refactoring solutions generated by our framework?* We conducted a pull-request study as mentioned earlier to show that our approach have practical value from the real-life developers' perspective.
4) We added formal definition of refactoring preconditions and the technical details of our refactoring framework, which are not available in [15] due to space limit.

The following of this paper is organized as follows. Section II introduces the background knowledge and motivation of this paper. Section III introduces an empirical study to manually refactor sub-classing by mocking. It lays the foundation of the proposed auto-refactoring framework. Section III-C formally defines the refactoring problem that replaces sub-classing by mocking based on the empirical study. Section IV introduces the implementation of the auto-refactoring framework and tool. Section V describes the research questions and evaluation rationale. Section VI discusses the quantitative evaluation results (RQ1-RQ4); and Section VII discusses the qualitative evaluation results (RQ5 and RQ6). Section VIII discusses the limitations of our approach. Section IX talks about related work. Section X concludes this paper.

II. BACKGROUND AND MOTIVATION

This section introduces the basic concepts of unit testing, and a motivating example comparing the difference between mocking through inheritance and through Mockito.

A. Unit Testing

Unit testing aims at validating that each unit of function performs as expected [1], [16]. The unit test code is composed of *test classes*, *test cases*, and *test suites*. A *test class* is similar to a production class. A *test class* contains one or more *test cases*. Each *test case* focuses on verifying the behavior of a certain unit of function (e.g. method) in the project. A *test case* should follow the "AAA (Arrange, Act, Assert)" pattern—arrange for setting up required test environment; act for invocation of the function being tested; and assert for checking whether the expectations were met [17]. A group of *test cases* for testing related functions are grouped and executed together as a *test suite*.

The interdependence among software units hinder our ability to perform unit testing. A key for creating high-quality, easy-to-maintain and debug unit test cases is to isolate the core *FUT* from its dependencies. In practice, this is achieved through mocking—replacing the dependency by a "faked" object.

B. A Motivating Example

Suppose, there is an e-Commerce system, which allows users to subscribe to its service. This function is achieved by a class named *CustomerService*. *CustomerService* defines a function, *subscribeCustomer*, to subscribe customers by email. The method, *subscribeCustomer*, depends on another class, *EmailManager*, which is responsible of managing and sending emails. Its method, *subscribe*, first sends an email to the customer to confirm the address; once confirmed, it stores the email address in a database. Another method, *sendEmail*, sends email through an external server. We aim to test the logic of *subscribeCustomer* in *CustomerService*. The problem is that, its dependency functions, *EmailManager*, is not fully implemented yet—neither the database nor the external service is available. Thus, we isolate the *FUT*, *subscribeCustomer*, from its dependency, *EmailManager*, by mocking the latter. Next, we illustrate mocking through inheritance and Mockito:

1) Mocking by Inheritance: Inheritance is a mechanism to derive a subclass from a base class. The subclass inherits the attributes and methods of the base class. Meanwhile, method overriding allows the subclass to replace certain method implementation of the base class. Inheritance is used as a "hand-rolled" approach for mocking. Developers define a test subclass to "mock" certain behaviors of the production class through method overriding or interface implementation for testing.

In Figure 1a, $MockEmailManager$ *extends* the $EmailManager$ (line 1). The former mocks the behaviors—$subscribe$ and $sendEmail$—of the latter through method overriding. Two new private attributes, $subscribed$ (line 2) and num (line 3), are defined for tracking the execution of the two overridden methods. That is, $subscribed$ is set to be *true* (line 6) when $subscribe$ executes; while num increments (line 10) each time $sendEmail$ executes. Of particular note, since the logic defined in this subclass prepares mocking behaviors for the unit test case, it is part of the "Arrange" in the "AAA" pattern.

The test case, $testSubscribeCustomer$, follows the "AAA" pattern. First, it arranges the environment for testing. This includes creating an instance of $MockEmailManager$—$emailManager$—(line 15), and creating an instance of $CustomerService$, $myservice$, which is the *FUT*. Next, it acts the *FUT* (line 17 and line 18). Lastly, the test case asserts the value of $subscribed$ and num with $MockEmailManager$ (line 19 and 20). They confirm that $subscribed$ is *true*, indicating method $subscribed$ is executed; and that num equals 2, indicating that two emails are sent (one asks the customer to confirm; the other sends a confirmation of subscription).

2) Mocking by Mockito: Mockito offers three aspects of capabilities for mocking. First, Mockito allows easy creation of a mock object as a "mock" or a "spy". The "mock" is a completely fake object and is entirely instrumented to track the interactions with it. In comparison, the "spy" wraps a real instance of the mocked object. The "spy" should be used when the execution of real methods is necessary in testing. Second, Mockito offers light-weighted method stubbing for controlling the behaviors of the mock object for testing purposes. Mockito provides dedicated syntax for different types of behavior—i.e. a void method, a return method, or a method for throwing exceptions. Third, Mockito provides explicit mechanism for verifying the behaviors/status of the mock objects. For instance, Mockito can ensure whether a mock method is being called or not, check on the number of calls made on a particular method, and take care of the order of calls, etc.

```
1  class MockEmailManager extends EmailManager {
2      private boolean subscribed = false;
3      private int num = 0;
4      @Override
5      public boolean subscribe()
6      {    subscribed = true;
7          sendEmail();
8          return true;}
9      @Override
10     public void sendEmail() {num++;}
11 }
12 class TestCustomerService {
13     @Test
14     public void testSubscribeCustomer() {
15         MockEmailManager emailManager = new MockEmailManager();
16         CustomerService myservice = new CustomerService();
17         myservice.subscribeCustomer(emailManager);
18         myservice.emailCustomers(emailManager);
19         assertTrue(MockEmailManager.subscribed);
20         assertEquals(2, MockEmailManager.num);}
21 }
```

Arrange / Subclass — Mock through method overriding

Arrange — Act — Assert

(a) Mocking by Inheritance

```
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 }
```

Arrange — Mock sendEmail — Act — Assert

(b) Mocking by Mockito

Fig. 1: A Motivating Example

In Figure 1b, Mockito directly creates a "mock" of the *EmailManager* (line 26), since the goal is to avoid its real execution and focus on its interactions with $subscribeCustomer$. In line 27-29, we stub the mocking behavior when $subscribe$ is invoked. The $sendEmail$ should *do nothing*, since we want to avoid sending real emails. Thus, there is no need to stub it. Acting the *FUT* (line 31 and line 32) remains the same as using inheritance. Finally, in line 33 and 34, we directly *verify* the execution of $subscribe$ and $sendEmail$.

3) Benefits of Mockito Over Inheritance: Mockito enables explicit and easy to understand testing logic. It allows easy creation of mock objects for different levels of function isolation (i.e. "mock" and "spy"). The *verify* functions in Mockito provide an explicit mechanism for checking the execution and status of the mock objects. In comparison, inheritance requires the developer to manually craft additional attributes/features in the subclass for tracking the execution of the mock objects. For example, new attributes, $subscribed$ and num, are used to keep track of method execution in the mock object. The logic behind the attributes is *implicit*, and may blur the testing logic.

Mockito decouples test and production code to ease the maintenance of the test code. Renaming methods/interfaces or reordering parameters in the production code will not break the test code, since Mockito wires the mock objects at run-time. In comparison, inheritance relationship increases the coupling between the test and production code. This unnecessarily cripples the inheritance hierarchy and increases maintenance difficulty. When the production code changes, its subclasses have to change accordingly.

Mockito improves the cohesion of test design by enforcing the "AAA" pattern of unit test case. Method stubbing through Mockito cohesively associates with the mock object when it is arranged in the test case. In comparison, in inheritance, the mock behavior (which is part of the "Arrange") is defined in a separate subclass through method overriding. It is detached from where the behavior is used for testing. This increases the cognitive load for understanding the test behavior.

III. EMPIRICAL STUDY

We first conduct an empirical study to investigate whether it is feasible and how to automatically replace inheritance by Mockito.

A. Dataset

We select *five* open source projects as our empirical study subjects—they are Dubbo [18], Druid [19], Accumulo [20], Cayenne [21], and CloudStack [22]. We select these projects because, first, they are popular open source projects from diverse problem domains. Second, test-production inheritance is common—each project contains 81 (CloudStack) to 291 (Druid) test subclasses for mocking. Thirdly, we are able to run the test cases in these projects, which is important for verifying the correctness of the manual refactoring. Most importantly, these projects already use *Mockito*.

B. Study Process

For each case where a test subclass inherits or implements a production class or interface, we investigate the following questions: *Can we manually refactor the inheritance by using Mockito based on our understanding? If so, is the refactoring process automatable? If not, what is the reason that makes the refactoring—and the automation—infeasible?* One author—the driver—manually reviews and refactors each test subclass, and the research team meets weekly to inspect and discuss the manual refactoring solutions:

1) If the manual refactoring is not feasible or not successful, the driver records detailed reasons.
2) For each refactored case, the driver summarizes the key refactoring steps, and determines whether the refactoring procedure can be automated. If automation is not possible, the driver records the reasons.
3) In the weekly meetings, the team: i) discuss and improve the manual refactoring solutions, and ii) discuss and define the auto-refactoring problem formalization.

C. Problem Formalization

Based on cases that are refactored in the empirical study, we formalize the auto-refactoring problem. It is a conversion from the left side to the right side:

$$Refactor(code_{inheritance}) \rightarrow code'_{mocking}.$$

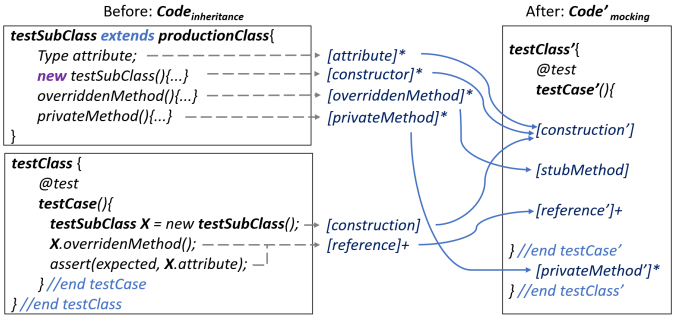


Fig. 2: Illustration of Refactoring

*a) **Before Refactoring**:* A refactoring candidate $code_{inheritance}$ can be abstracted as a triad:

$$code_{inheritance} = \\ < testSubClass, productionClass, testClass >$$

Here, $testSubClass$ extends the $productionClass$. The $testClass$ leverages $testSubClass$ to assist testing. The left-side code snippets in Figure 2 illustrates the formalization of $code_{inheritance}$.

The $testSubClass$ is further consisted of four key elements (The convention "[]*" indicates that there is zero or more of a design element). The upper-left code snippet illustrates an example *testSubClass*.

$$testSubClass =< [constructor]*, [attribute]*, \\ [overriddenMethod]*, [privateMethod]* >$$

- *constructor* creates a $testSubClass$ instance.
- *attribute* is for tracking the execution of $testSubClass$.
- *overriddenMethod* defines dummy implementation of a function in $productionClass$.
- *privateMethod* defines additional function in $testSubClass$.

A $testClass$ leverages the $testSubClass$ for testing, which can be formalized as following:

$$testClass =< [testCase]+ >$$
$$testCase =< [construction], [reference]+ >$$

A $testClass$ contains at least one $testCase$. A $testCase$ involves a $testSubClass$ in two parts for fulfilling the testing goal: First, *construction*, which invokes a *constructor* of $testSubClass$ to create an instance. Second, *reference*, which accesses the attributes or call the methods of the instance. The lower-left code snippet of Figure 2 illustrates a simple example.

*b) **After Refactoring**:* The original $code_{inheritance}$ is refactored into $code'_{mocking}$, which eliminates $testSubClass$ and replaces it by a mock object:

$$code'_{mocking} =< productionClass, testClass' >$$

Thus $testClass$ becomes $testClass'$, and each $testCase$ in it becomes $testCase'$:

$$testClass' =< [testCase']+, [privateMethod']* >$$
$$testCase' =$$
$$< [construction'], [stubMethod]*, [reference']+ >$$

As illustrated in the right-side code snippet in Figure 2, *testClass′* is composed of *testCase′* and *[privateMethod′]*. The *[privateMethod′]* is the *[privateMethod]* moved from *testSubClass* to *testClass′*. And, each refactored *testCase'* is consisted of three components. First, *[construction′]* to create a mock object of the *productionClass*, which replaces the instance created by *[construction]* in *testCase*. Second, *[stubMethod]*, which replaces the *[overridenMethod]* in *testSubClass*. Third, *[reference′]* to the mock object, which replaces the respective *[reference]* to the *testSubClass* instance in *testCase*. We will explain the formal refactoring procedure in Section IV-A.

c) Refactoring Pre-conditions: Based on the empirical study, we summarize ten refactoring pre-conditions. For any refactoring candidate, denoted as $Code_{inheritance} =< testSubClass, productionClass, testClass >$, the three elements and/or their relations must satisfy certain pre-conditions to match the refactoring formalization defined above. Otherwise, it is infeasible to refactor the $Code_{inheritance}$ instance or the refactoring requires case-by-case effort and cannot follow any automated process. Following, we formally define each pre-condition, in the notion of the three elements in $Code_{inheritance}$—namely $testSubClass$, $productionClass$, and $testClass$ to specify the characteristics and relations that must be met, and provide the rationale behind each pre-condition:

1) The $testSubClass$ *implements* or *extends* one and only one $productionClass$.

$$onlyOneSuperClass \equiv$$
$$\exists! productionClass \mid$$
$$extend(testSubClass, productionClass)$$
$$\vee\, implement(testSubClass, productionClass)$$

Rationale: It is possible that $testSubClass$ extends a parent class, and implement multiple other classes. If this happens, it implies that $testSubClass$ may mock the behavior of multiple production interfaces. However, in Mockito, mocking multiple $productionClasses$ is not recommended [23]. If we force the refactoring, we need to use extraInterfaces() to bind to multiple interfaces. In the method stubbing, we need to cast the mock object to the respective interface type when calling the Mockito.when(). Mockito generally does not recommend such usage in their official documentation *"This mysterious feature should be used very occasionally. The object under test should know exactly its collaborators dependencies. If you happen to use it often than please make sure you are really producing simple, clean readable code."* [1]. Our initial motivation behind refactoring is to improve the design and understanding of test cases by using Mockito, but such refactoring may contradict our initial goal. This pre-condition should apply regardless of the choice of mocking framework as mocking multiple production interfaces with one mock object generally complicates the design and hinders comprehension.

2) The $testSubClass$ does not *override* two JDK APIs, i.e. equals() and hashCode().

$$doNotOverrideJDK \equiv$$
$$\nexists\, method \in\ testSubClass \mid$$
$$override(method, java.lang.Object.equals())$$
$$\vee\, override(method, java.lang.Object.hashCode())$$

Rationale: A general design principle in using mocking is that *"Only mock types you own"* [12]. Thus, developers should generally avoid mocking JDK APIs which are not owned by their project. *Mockito* is built upon the two JDK APIs, namely *java.lang.Object.equals()* and *java.lang.Object.hashCode()* [24]. In addition, mocking the behavior of these two APIs will endanger the normal functions of Mockito. As Mockito states in their WiKi page that *"Mockito Cannot mock equals(), hashCode(). Firstly, you should not mock those methods. Secondly, Mockito defines and depends upon a specific implementation of these methods. Redefining them might break Mockito."*[2]. Even other mocking frameworks may not be built on these two APIs in specific, this pre-condition should still apply due to the general principle of *"Only mock types you own"*.

3) The $testSubclass$ does not contain any new public method that is not defined in its parent $productionClass$, and, this method is used somewhere else other than $testSubclass$:

$$noNewPublicMethod \equiv$$
$$\nexists public\ method \in testSubclass \mid$$
$$method \notin productionClass$$
$$\wedge\, method\ used\ outside\ testSubclass$$

Rationale: If $testSubClass$ contains a *public* method that is not defined in the $productionClass$, and this new method is used somewhere by $testClass$, it implies that $testSubClass$ contains extra, new behaviors compared to $productionClass$. In this case, $testSubClass$ with the extra function is no longer simply "mocking" $productionClass$. To our best knowledge, no existing mocking framework supports adding new methods to the mocked object. So this pre-condition applies regardless of the choice of the mocking framework.

4) The $testSubclass$ does not have self reference.

[1] https://javadoc.io/static/org.mockito/mockito-core/3.1.0/org/mockito/MockSettings.html#extraInterfaces-java.lang.Class...-

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

$$noSelfReference(testSubClass) \equiv$$
$$\nexists element \in testSubClass \mid$$
$$instance\_of(element, testSublass)$$

Rationale: If $testSubClass$ contains self-reference, such as containing an attribute in its own type, or using a parameter or variable in its own type in methods, etc., it means that the mock object's behavior depends on itself. When creating a mock object and defining method stubbing using Mockito (or even with another mocking framework), we cannot properly translate the logic of self-reference to its equivalent form, since self-reference is not supported. To our best knowledge, no existing mocking framework supports mock objects with self-reference. Thus, this principle should apply regardless of the choice of mocking framework.

5) The $testSubclass$ is instantiated through its constructor.

$$isInstantiatedByConstructor \equiv$$
$$\exists X, \mid X\ calls\ testSubclass.constructor()$$

Rationale: If $testSubClass$ is never instantiated anywhere, in X, which is a $testClass$ or $testCase$ through its constructor, it implies that either it is not used anywhere or is instantiated through dynamic binding. In the former case, it means that the mock object is not used anywhere, and thus there is no need for a refactoring. If it is the latter, we cannot use Mockito, or another mocking framework, to work with dynamic binding. This precondition should apply regardless of the choice of the mocking framework.

6) The $testSubClass$ does not contain special code annotation.

$$noSpecialAnnotation \equiv$$
$$element \in\ testSubClass \mid$$
$$element.annotation == null$$
$$\vee\ element.annotation \in$$
$$\{\text{“}SuppressWarning\text{”}, \text{“}Override\text{”}\}$$

Rationale: Some $testSubClass$ contains highly customized and project specific annotations or annotations from a special library [25]. These special annotations are not supported by Mockito, or any other existing mocking framework. Thus, this precondition applies not only to Mockito. Our approach only accepts two common annotations, including *SuppressWarning* and *Override*, based on our empirical study observations. We acknowledge that only including these two annotations observed from our empirical study may post limitation to our approach for its general applicability. We will discuss this more in Section VIII.

7) The $testSubclass$ does not access to a protected attribute or method in the $productionClass$.

$$noAccess2Protected \equiv$$
$$\nexists protected\ element \in\ productionClass \mid$$
$$reference(testSubclass, element)$$

Rationale: *Mockito* does not support the access to protected elements. This is a limitation with Mockito. One could use a more powerful mocking framework, such as PowerMock [7] to overcome this limitation. PowerMock supports accessing to protected fields and methods through reflection [26].

8) If a $testSubclass$ contains a new attribute—i.e. the attribute is not defined in its parent $productionClass$, no such a $testSubclass$ instance is declared in one method, namely $method_i$, and passed to another method, namely $method_j$, which uses its new attribute.

$$noPassingWithNewElement \equiv$$
$$\exists attribute \in testSubclass$$
$$\wedge\ attribute \notin productionClass \mid$$
$$\nexists X \mid instance\_of(X, testSubclass)$$
$$\wedge\ X\ declared\ in\ method_i \in testClass$$
$$\wedge\ X.attribute\ used\ in\ method_j \in testClass$$

Rationale: The refactoring requires a case by case understanding of how $X.attribute$ is used in $method_j \in testClass$, as well as whether and how $method_j \in testClass$ is used by other methods in $testClass$ for a clean design. If without a thorough understanding of how the attribute is used across methods, one may just refactor the case by passing among methods the new mock object together with the related variable (the variable is for replacing the attribute in the sub-class). Although this may provide equivalent logic, the design looks messier due to passing multiple parameters. Thus, it requires manual effort to create a good test design based on case by case understanding of the new attribute. This principle is independent from the choice of mocking framework, as refactoring such cases prudently tends to compromise the test design due to the complexity.

9) If the $testSubclass$ contains a new attribute—i.e. the attribute is not defined in its parent $productionClass$, no such a $testSubclass$ is created and used as a collection of its instances in $testClass$.

$$notBatchUse \equiv$$
$$\exists attribute \in testClass$$
$$\wedge\ attribute \notin productionClass \mid$$
$$\nexists declaration = new\ Class_i < testSubclass >$$
$$\wedge\ Class_i\ implements\ Iterable$$

Rationale: If $testSubclass$ contains a new attribute and is created as a collection of instances, i.e. as *Iterable*, we cannot properly preserve the logic of the new attribute of $testSubClass$ in the refactoring. More specifically, we could create a collection of variables, which track the new attribute, together with a collection of mock objects, to replace the $testSubClass$ instance collection. But the logic for binding each variable and each mock object from the two collections could go far more complicated

compared to the one single collection of $testSubClass$. Thus the refactoring would not merit good design and easy comprehension of test cases. This pre-condition should apply regardless of the choice of mocking framework due to the same rationale as the previous pre-condition.

10) The $testSubclass$ does not contain the definition of an inner class.

$$noInnerClass \equiv$$
$$\nexists class \mid class.definition \in testSubclass$$

Rationale: The inner class is defined such that it is only used inside of the $testSubclass$ to assist the logic of mocking behavior. To enable the refactoring of such cases, the developer should first use an existing refactoring tool with IDEs, which focus on moving inner class to its containing class (e.g. *Move Inner to Upper Level Dialog* [3] in IntelliJ, and *Convert Member Type to Top Level* [4] in Eclipse); then use our tool (also as an IDE Plugin) as the follow-up step to automatically refactor the $testSubclass$. Again, this principle is independent from the choice of mocking framework, and should apply generally.

Of a particular note, our discussion of the rationale behind each pre-condition is detailed based on refactoring using Mockito. As mentioned earlier, we choose Mockito due to its wide adoption. We would like to point out that the pre-condition #7 could be potentially loosen up if using a more powerful framework, i.e. PowerMock. However, the feasibility should be rigorously tested, which is out of the scope of this study, and we leave it for the future.

The remaining nine pre-conditions should generally apply regardless of the choice of the mocking framework. This happens in two scenarios. First, to our best knowledge, existing mocking frameworks suffer from the same constraints as Mockito, and thus the same pre-conditions should apply. These include #2 ("only mock types you own"), #3 (cannot support new methods with mock objects), #4 (cannot support mock objects with self-reference), #5 (cannot support dynamic binding for creating mock objects), and #6 (cannot support special annotations). The second scenario is that some pre-conditions identify certain design features that should not favor the refactoring, even though refactoring using Mockito or another framework is possible. These include #1, #8, #9, and #10.

D. Findings and Implications

Table I shows the detailed findings of the empirical dataset. The first column lists the project names. The second column lists the total number of test sub-classes, which is the initial refactoring candidate pool we identify before checking the ten pre-conditions. As we can see, we start from a total of 832 test sub-classes for the investigation.

The following columns, "P1" to "P10", list the number of test sub-classes that are excluded because they do not meet the respective refactoring precondition. As shown in Table I, the top three commonly violated pre-conditions are "P5", "P6", and "P3", which exclude 16%, 12%, and 8% cases, which add up to 36% of the excluded cases. More specifically, as introduced earlier, "P5" requires that the test sub-class must be instantiated somewhere through its constructor. There are two possibilities if this pre-condition is not met. First, the test subclass becomes deprecated and never used, thus no need to create a mock object using Mockito to replace it. Or, second, the test subclass is instantiated through dynamic binding—for this, we cannot use Mockito to create a mock object through dynamic binding. "P6" requires that the test sub-classes do not contain special annotations since Mockito does not support them. It is an interesting future direction to advance the capability of mocking frameworks to support special code annotations. Finally, "P3" requires that the test sub-class does not include new public method that used outside of itself. This indicates that the test subclass contains new, extra behavior that is not in the object being mocked. Thus the test subclass is no longer simply "mocking" its parent, and we should not replace it by a mock object using Mockito. This could be a scenario where a test sub-class is a better design than using Mockito due to the complexity of the behavior.

The second last column shows the number of test sub-classes that meet all pre-conditions but failed refactoring due to execution issues. In these cases, we either have issues configuring the projects and executing related test cases; or the test behavior changes after refactoring for reasons that require case-by-case investigation. A total of 97 (12%) cases failed the refactoring due to execution issues.

The last column shows the ultimate number of test sub-classes that we are able to refactor successfully. A total of 222 (27%) cases can be refactored successfully, and the refactoring can be potentially automated. This non-trivial portion of cases motivate the design of our automated refactoring framework. Later, we use our implemented refactoring framework (Section IV) on the empirical dataset, and confirmed that these 222 cases can be refactored by our tool automatically and successfully.

IV. REFACTORING FRAMEWORK

The auto-refactoring framework, implemented as an Eclipse-plugin [5], addresses the above formalization with two parts. The first part is identifying refactoring candidates. After loading a project in Eclipse, a user first selects the scope, e.g. the entire project, a package, or a group of files, from which refactoring candidates should be identified. The identification relies on the *AST Parser* of *Eclipse JDT* [27] to detect feasible candidates that meets refactoring pre-conditions defined in Section III-C . The second part is refactoring each candidate. The tool will notify users the list of identified refactoring candidates (i.e. sub-classes). The user needs to select a candidate to proceed

[3] https://www.jetbrains.com/help/idea/move-inner-to-upper-level-dialog-for-java.html

[4] See the first response for https://stackoverflow.com/questions/2117962/how-to-refactor-a-static-inner-class-to-a-top-level-class-in-eclipse

[5] https://github.com/wx930910/JMocker

TABLE I: Manual Refactoring Pre-Conditions details

Proj.	#Subcl.	# Cases Excluded By Pre-Conditions										Exec. Iss.	Succ.
		P1	P2	P3	P4	P5	P6	P7	P8	P9	P10		
Dubbo	148	9 (6%)	1 (1%)	0 (0%)	12 (8%)	39 (26%)	5 (3%)	10 (7%)	11 (7%)	0 (0%)	1 (1%)	14 (9%)	46 (31%)
Druid	291	7 (2%)	5 (2%)	40 (14%)	23 (8%)	6 (2%)	54 (19%)	11 (4%)	10 (3%)	2 (1%)	0 (0%)	60 (21%)	73 (25%)
Accumulo	161	1 (1%)	1 (1%)	15 (9%)	10 (6%)	42 (26%)	7 (4%)	6 (4%)	20 (12%)	1 (1%)	0 (0%)	11 (7%)	47 (29%)
Cayenne	151	2 (1%)	1 (1%)	10 (7%)	14 (9%)	47 (31%)	12 (8%)	11 (7%)	10 (7%)	0 (0%)	0 (0%)	2 (1%)	42 (28%)
CloudStack	81	26 (32%)	0 (0%)	0 (0%)	1 (1%)	3 (4%)	22 (27%)	2 (2%)	3 (4%)	0 (0%)	0 (0%)	10 (12%)	14 (17%)
Sum	832	45 (5%)	8 (1%)	65 (8%)	60 (7%)	137 (16%)	100 (12%)	40 (5%)	54 (6%)	3 (0%)	1 (0%)	97 (12%)	222 (27%)

with the refactoring. The implementation of refactoring relies on the *ASTRewrite* mechanism of the *Eclipse JDT* [27].

The identification of feasible refactoring candidates simplify matches the ten pre-conditions defined in III-C. This section focuses on the introduction of the details refactoring procedure once a candidate is selected for refactoring.

A. Auto-Refactoring Procedure

Figure 3 shows the refactoring procedure to convert $code_{inheritance}$ to $code'_{mocking}$ for each refactoring candidate identified from the previous step. Our approach involves five logical parts:

1) Create mock object: This step constructs a mock object using Mockito to replace the $testSubclass$ instance, and ensures that they have equivalent initial status.
2) Preserve mocking behavior: This step extracts the overridden methods and moves the private methods in $testSubClass$ to ensure that the mock object has equivalent behavior as the $testSubClass$ instance.
3) Preserve references to the mock object: This step ensures that the execution/verification of the mock object is equivalent to that of the $testSubClass$ instance.
4) Infrastructure Procedure-*translateToMocking*: This procedure cross-cuts the three previous parts to ensure that the refactoring follows the mocking syntax.
5) Create *MockMethod* for code reusability: This applies when multiple test cases could reuse the mock object creation.

In the following subsections, we will explain each part in detail.

1) Step1-Create Mock Object: This step creates a mock object using Mockito to replace the $testSubClass$ instance. To ensure that the initial status of the $testSubClass$ instance and the mock object are equivalent, the following three substeps are performed:

Step-1.1: Replace $testSubClass$ instance creation by mock object creation. There are two ways to do so—through *spy* or *mock*, as illustrated in Figure 4a and Figure 4b, respectively. *Spy* creates a real object; while *mock* creates a complete mock or fake object. Based on the empirical study, if the *productionClass* is an interface without any method definition, we should use *mock*, since an interface cannot be instantiated as a real object. In comparison, if the *productionClass* has method implementation, we should use *spy* to ensure that the mock object has the same behavior as the real object, except for the purposely stubbed methods. There are other minor syntax variations for *spy* and *mock*, summarized here [6].

[6]https://sites.google.com/view/mockrefactoring

Step-1.2: Extract the $attributes$ of $testSubClass$ to $testClass'$. This ensures that the status of the $testSubClass$ instance is preserved for the mock object. We observed two types of $testSubClass$ attributes from the empirical study, which are treated differently.

The first type of attribute is the "*counter/checker*" as shown in the motivating example in Figure 1. These attributes are for tracking the execution of the mock object. We recognize the type using three heuristics. First, it is a *boolean* or an *int*. Second, it is only read/written in a certain methods of the mock object. Third, it is asserted for checking the execution of the associated methods. Mockito has a designated mechanism—*Mockito.verify*—for verifying its execution. Thus, there is no need to preserve this type of attributes. Instead, we just keep a record of the tracked methods and verify their execution later using *Mockito.verify* to replace the assertions.

The other type of attributes, which may also serve for tracking related execution information, are in diverse types, and could be referenced anywhere in $testClass$. The way that we extract such an attribute from the $testSubClass$ to $testClass'$ depends on how $testSubClass$ is originally used in $testClass$. More specifically, if the $testSubClass$ instance is an attribute of the $testClass$, the attribute of $testSubClass$ will become an attribute for $testClass'$, to ensure the same access scope. Otherwise, if the $testSubClass$ instance is created as a local variable inside a $testCase$, the attribute of $testSubClass$ will become a local variable in $testCase'$.

Step-1.3: Extract the constructor logic from $testSubClass$ to $[construction']$. This ensures that the mock object has equivalent initial status as the $testSubClass$ instance. If the $testSubClass$ instance is created using a default constructor, this step can be skipped. If $testSubClass$ instance is created using a non-default constructor (which comes with additional settings for the created instance), the constructor logic needs to be extracted to $[construction']$. Each statement in the constructor needs to be translated to follow the syntax after the refactoring. Here, an infrastructure procedure named *translateToMocking* takes the code body of the constructor as input, and translates each statement following the mocking syntax. Since *translateToMocking* cross-cuts all three logic steps of the refactoring procedure, we will introduce its details in Section IV-A4.

2) Step2-Preserve Mocking Behavior: This preserves the mocking behaviors by treating the *overriddenMethods* and *private-Methods* in the $testSubClass$:

Step-2.1: Extract the $overriddenMethod$ in $testSubClass$ to the $stubMethod$ which directly associates with the mock object created/used in $testCase'$. There are two common ways to stub a method: *doAnswer* and *thenAnswer*. The *thenAnswer*

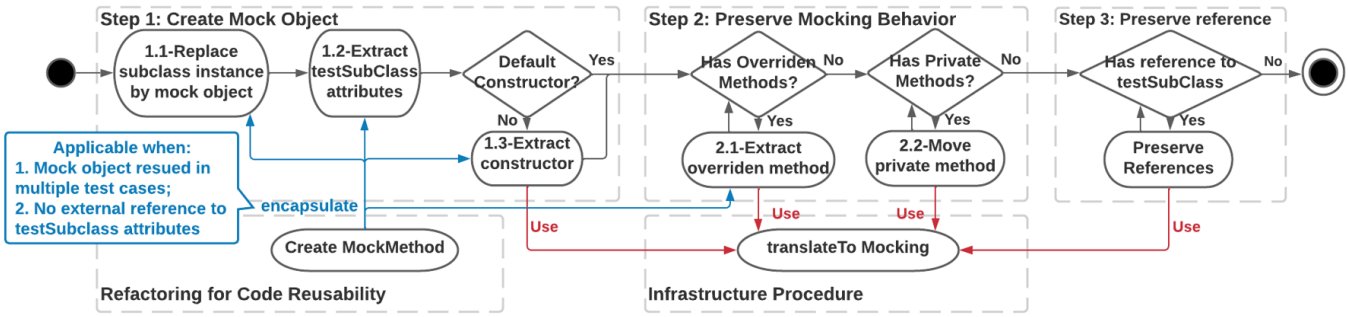
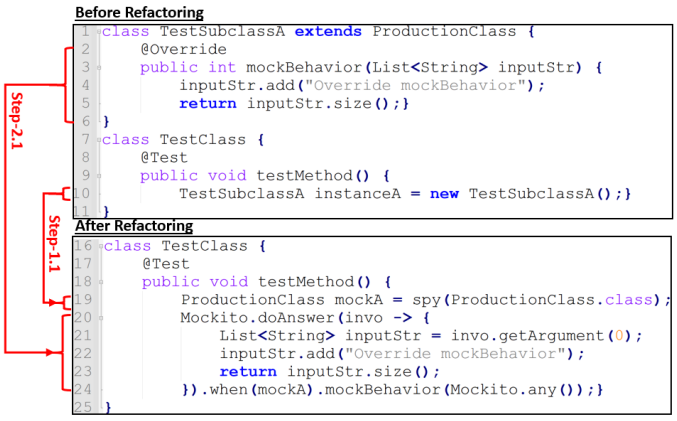
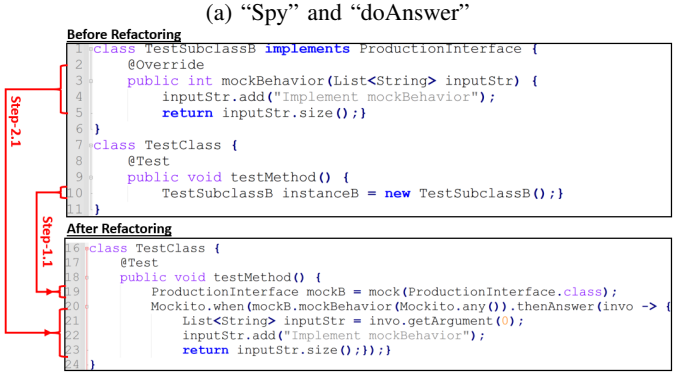


Fig. 3: Automated Refactoring Procedure

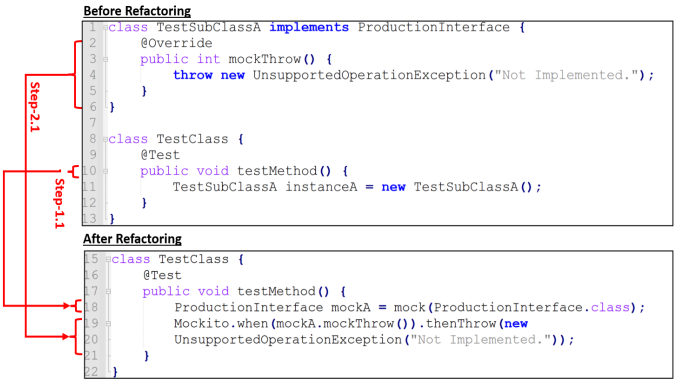


(a) "Spy" and "doAnswer"



(b) "Mock" and "thenAnswer"

Fig. 4: Mock Object Creation and Stub Method



(a) thenThrow Method Stub

(b) doThrow Method Stub

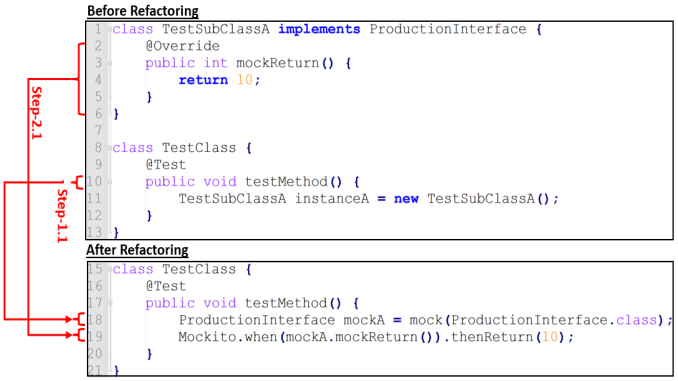
Fig. 5: Throw Method Stub

adds **additional** actions to the stubbed method [28]. It ensures *type safety* thus should be preferred whenever possible. While, *doAnswer* entirely replaces the original method behavior [29], working similar to method overridden in inheritance. Based on empirical experience, *thenAnswer* works with objects created using *mock*; while the *spy* object should work with *doAnswer* to preserve the "overridden" behavior. Figure 4 illustrates *doAnswer* in Figure 4a (line 20 to line 24) and *thenAsnwer* in Figure 4b (line 20 to line 23) respectively. They are used to replace the overridden methods between line 2 to line 7 in Figure 4a and in Figure 4b.

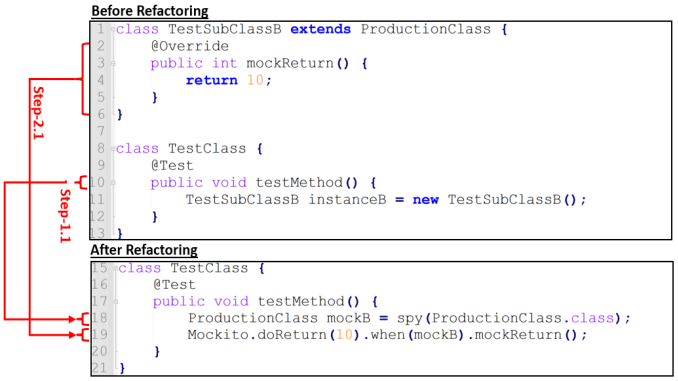
Note that the internal logic of the overridden methods in Figure 4 is straightforward —i.e. without referencing attributes or methods in the $testSubclass$. Thus we can directly move them to the stub method blocks. If the internal logic has a reference to the $testSubClass$ attributes/methods, we also need to use *translateToMocking* procedure to convert the syntax before moving.

In addition, there are other method-stubbing syntax for different kinds of behaviors, including *thenThrow* vs. *doThrow* (Figure 5) for throwing exceptions, *thenReturn* vs. *doReturn* (Figure 6) for returning values, and *doNothing* (Figure 7) for void behavior.

More specifically, *thenThrow* and *doThrow* are used when the overridden method throws an exception (line 4 in Figure 5a and in Figure 5b). The exception-throwing in *TestSubClassB* becomes method stub in $testCase'$ (line 19 in Figure 5a and in Figure 5b). The difference is that *thenThrow* goes through the original method's behavior first and then throws the exception; it cannot be used when the overridden method return *void*. In



(a) thenReturn Method Stub



(b) doReturn Method Stub

Fig. 6: Return Method Stub

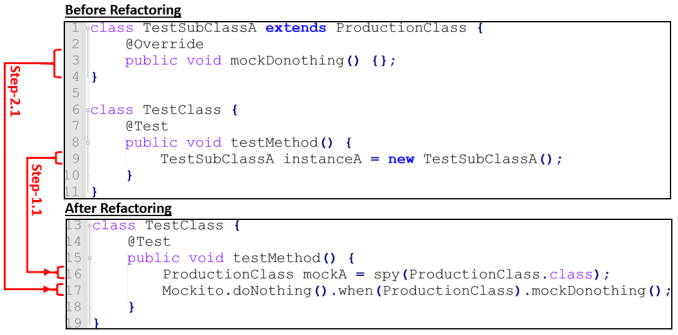


Fig. 7: doNothing Method Stub

comparison, *doThrow* completely ignores the original method behavior, and throws the exception. We use *thenThrow* for *mocking object* and use *doThrow* for *spying object*.

Similarly, *thenReturn* and *doReturn* are used when the overridden method returns certain objects (line 4 in Figure 6a and in Figure 6b). The return behavior in *TestSubClassB* is handled by method stub in $testCase'$ (line 19 in Figure 6a and in Figure 6b). We use *thenReturn* for *mocking object* and *doReturn* for *spying object*.

Finally, *doNothing* is used when no action is needed. The overridden method $mockDonothing()$ (line 3 in Figure 7) is replaced by *doNothing* (line 17 in Figure 7). *doNothing* is only used for *spying object* to completely skip the original behavior. *Mocking object* will do nothing by default and thus there is no need to define the respective stub method.

Step-2.2: Move each private method from $testSubClass$ to *testClass*′. These method movements cannot be directly copy-and-pasted due to the overall syntax change. Similarly, we use the *translateToMocking* procedure to convert the method syntax when moving it. In addition, the method signature may need to be updated accordingly, to take additional input parameters, for accessing the local variables in $testCase'$ which were the attributes in $testSubClass$.

3) Step3-Preserve Reference to the Mock Object: In a $testCase$, there could be references to the attributes and/or methods of the $testSubClass$ instance—as such the instance is created for facilitating testing. To ensure that the behavior of $testCase$ and $testCase'$ remains consistent, we need to preserve these references on the mock object. Again, we use the *translateToMocking* procedure to preserve $[reference]*$ in $testCase$ to be the respective $[reference']*$ in $testCase'$.

4) Infrastructure Procedure—translateToMocking: As mentioned earlier, each previous step relies on the *translateToMocking* procedure which takes a certain code body in the $testSubClass$—e.g. methods, constructors, reference statements—as input, and convert them to follow the syntax after refactoring. Figure 8 shows the pseudo-code of this procedure.

Algorithm 1 translateToMocking (codeBody)

Require: TSC = testSubclass
Require: TCL = testClass
Require: TCA = testCase
Require: MO = mockingObject

```
1:  while codeBody has stm do
2:    if stm.contains(TSC.attr) then
3:      if TSC.attr is "checker/counter" then
4:        if stm is "assertion" then
            stm.replace(assertion, MockVerify)
5:        else
            error!
6:        end if
7:      else
8:        if TSC.attr becomes TCA.variable then
            stm.replace(TSC.attr, TCA.var)
9:        end if
10:       if TSC.attr becomes TCL.attr then
            stm.replace(TSC.attr, TCL.attr)
11:       end if
12:     end if
13:   end if
14:   if stm.contains(TSC.privateMethod) then
        stm.replace(TSC.privateMethod, TCL.privateMethod)
15:   end if
16:   if stm.contains(TSC.overriddenMethod) then
        stm.replace(TSC.overriddenMethod, MO.stubMethod)
17:   end if
18:   if stm.contains(TSC.setter/getterMethod) then
        stm.inline(translateToMock(TSC.setter/getterMethod.codeBody))
19:   end if
20: end while
```

Fig. 8: translateToMocking pseudo-code

For each statement, stm, in the input code body, this procedure makes the following conversions: If stm has a reference to an attribute in $testSubclass$ (line 2-13), it is treated in two different ways depending on the attribute type. First, if the attribute is a *"checker/counter"*, we just remove stm, since there is no need to keep track of this attribute anymore. The respective assertion statements, where the

attribute is checked, are replaced by the *MockVerify* statements of the associated methods (line 4). Second, if the attribute is a general type (i.e. other than a *"checker/counter"*), we just replace the attribute in stm by the respective local variable in $testCase'$ (line 8-9)—or the attribute in the $testClasss'$ (line 10-11)—depending on where the attribute is extracted in step 1.2. If stm contains reference to a $privateMethod$ in the $testSubClass$, we replace the reference to be the $privateMethod'$ in $testClass'$ (line 14). Similarly, if stm contains reference to an $overriddenMethod$ in $testSubClass$, we replace this reference by the *stub method* (created in step 2.1) associated with the mock object in $testCase'$ (line 16). If stm contains reference to $setterMethod$ or $getterMethod$, we just make the method inline with where it is used, since a setter/getter method is usually one line of code.

5) Create MockMethod for Code Reusability: A *testSubClass* could be created and used in multiple $testCases$. For each constructor in $testSubClass$, the respective $[construction']$ block after refactoring—generated by Step 1.1, 1.2, and 1.3, as well as all the $[stubMethod]$ blocks—generated by Step 2.1—can be reused whenever this constructor is called. To prevent code-clone in such cases, we encapsulate these blocks within a separate *MockMethod* in the $testClass$ for reuse. However, the *MockMethod* is not appropriate when there exists external reference to the $testSubClass$'s attributes in the $testCases$. The external reference to the $testSubClass$ attributes cannot be preserved, since the attributes become the local variables in the *MockMethod*. Thus, the condition to apply *MockMethod* includes: 1) the mock object is reused in multiple $testCases'$; and 2) there was no reference to the $testSubClass$'s attributes before refactoring.

V. Research Questions and Evaluation Design

A. Research Questions

We aim to evaluate our approach by six RQs. RQ1 to RQ4 are answered by quantitative evaluation. RQ5 and RQ6 are answered by qualitative evaluation from real software developers and open source projects.

- *RQ1: How common are test sub-classes for mocking in Apache projects? And what percentage of them qualify as refactoring candidates for our approach?* On the one hand, if there are only a few test sub-classes found on Apache projects, it indicates that our approach does not have many practical use cases. On the other hand, if only a small portion of the test sub-classes meet the refactoring pre-conditions, it indicates that our approach does not find sufficient refactoring candidates to provide practical value.
- *RQ2: What percentage of the refactoring candidates can actually be successfully refactored using our approach?* Following RQ1, the goal is to investigate whether the refactoring framework can be successfully applied to new projects, and what is the percentage of refactoring candidates that can be actually successfully refactored by our approach.
- *RQ3: Do the test behaviors remain consistent before and after the refactoring with injected mutations?* Mutation testing is a proxy for evaluating the behaviour preservation of the refactored test cases in terms of detecting potential defects. We use mutation testing to inject potential defects, as mutants, into the production code. Behavior preservation means that the same mutants should be executed consistently by the test cases before and after the refactoring. Furthermore, they should be killed (i.e. failing the test cases) or survive (i.e. passing the test cases) consistently by the test cases that execute them before and after the refactoring.
- *RQ4: How does the refactoring affect the size of and coupling within the code base?* Increased code base size and coupling are associated with higher maintenance cost. If the code base size or code coupling increase after the refactoring, our approach does not provide practical value. We argue that one of the main benefits of our refactoring is to decrease the decoupling between the test code and production code. Of a particular note, this RQ does not consider a classic code complexity metric, called cyclomatic complexity, since this metric counts the number of decisions, e.g. *if-else* branches, based on the flow graph of the source code. Our refactoring does not change the decision flow of the code, thus the cyclomatic complexity is not be impacted.
- *RQ5: What is the performance of our refactoring framework?* If the refactoring framework takes a long time to execute, it is not affordable in practice.
- *RQ6: How is the quality of our auto-refactoring solutions in real-developers' opinion? And how does it compare to the manual refactoring solution implemented by developers?* We aim to understand the value, benefits, and quality of our refactoring solutions by taking real developers opinion, especially when compared to manual refactoring solutions by developers. For this purpose, we conduct a user study involving full-time real developers to let them both manually implement the refactoring and review the refactoring solutions generated by our framework. We will explain the detailed user study design in Section V-C
- *RQ7: How well do the open source projects receive the refactoring solutions generated by our framework?* We aim to understand whether the auto-generated refactoring solutions are acceptable in practice by the open source projects. If not, what are the gaps? This helps us identify future research opportunities. For this aim, we submit at least one pull request with the auto-generated refactoring solution to each project. We will explain the detailed study design of RQ7 in Section V-C

In the following, we explain the detailed evaluation methods and rationale.

B. Evaluation Dataset

There are a total of 241 Java projects on the Apache Software Foundation. Among these, only 182 projects contain test files. These 182 projects form the scope of RQ1. If a project does

not contain any test code in the first place, it is meaningless to investigate it in terms of whether it uses inheritance for mocking. Table II shows the basic information of these 182 projects—including the average (column Avg.), median (column Med.), maximal (column Max.), and minimal (column Min.) numbers of Java files (1st row), test files (2nd row), and LOC (3rd row). These projects contain on average 1,290 Java files, and 408 test files, with 143K LOC. In RQ1, we focus on investigating the number and percentage of feasible refactoring candidates we can identify in these 182 projects. This helps us to understand the general applicability of our approach.

TABLE II: Basic Information of Apache Java Projects

Basic Information	Avg.	Med.	Max.	Min.
#Java Files	1,290	734	7,982	14
#Test Files	408	191	3,604	1
LOC	143,059	73,923	1,351,942	554

Starting from RQ2, we rely on nine projects that are independent from the projects used in the empirical study. Note that we cannot afford to investigate RQ2 to RQ7 on the 182 Apache projects, since it requires manual configuration of each project, manual execution and comparison of each test case before and after the refactor to make sure that the refactoring is actually successful (RQ2) and the behavior preserves (RQ3).

The nine projects, with a total of 774 test subclasses, are: JackRabbit—an open source content repository for the Java platform [30], Log4J2—a Java-based logging utility [31], Qpid-Proton-J—a high-performance, lightweight messaging library [32], Apache Commons—which focuses on all aspects of reusable Java Components, with 40 subprojects, including Commons-Collections, Commons-Lang, Commons-Logging, etc, Sakai—an open-source learning management system [33], Curator, a Java/JVM client library for Zookeeper [34], Avro—a data serialization system [35], PDFBox—an open-source library for working with PDF documents [36], OpenNLP—a machine learning based toolkit for the processing of natural language text. To avoid bias, we intentionally select these projects since their domains differ from the training dataset. The rationale of other selection criteria is similar to that of the training dataset in Section III.

Table III lists the detailed information of the evaluation dataset. Column 2 shows the number of production files in each project, ranging from 358 (Qpid-Proton-J) to 5,673 (Sakai). Column 3 shows the number of test files, ranging from 105 (Qpid-Proton-J) to 2,820 (Commons). Of a particular note, Column 4 shows the number of test files that use inheritance for mocking purposes; while column 5 shows the number of test files that uses a mocking framework. On average, 7% of the test files use inheritance for mocking; in comparison, 5% of test files use a mocking framework. This suggests that the use of inheritance is common for mocking—comparable to the use of a mocking framework. Therefore, our approach can benefit these projects.

C. Evaluation Design and Rationale

1) RQ1: For the first part of RQ1, we collect the total number of test classes in the 182 Apache projects, and count

TABLE III: Evaluation Dataset

Project	#P. Files	#T. Files	#Files Use IM (%)	#Files Use MF (%)
JackRabbit	2,107	1,021	27 (3%)	18 (2%)
Log4J2	2,031	864	158 (18%)	43 (5%)
Qpid-Proton-J	358	105	3 (3%)	20 (19%)
Commons	4,888	2,820	216 (8%)	111 (4%)
Sakai	5,673	494	18 (4%)	128 (26%)
Curator	518	172	10 (6%)	14 (8%)
Avro	421	830	9 (1%)	0 (0%)
PDFBox	1,089	196	1 (1%)	0 (0%)
OpenNLP	734	234	61 (26%)	15 (6%)
Total	17,819	6,736	503 (7%)	355 (5%)

the number of test sub-classes for mocking—i.e. test classes that inherit a production class or implements an interface in the production code. If there is only a small number of test sub-classes for mocking in Apache projects, it implies that our approach lacks practical use cases.

For the second part of RQ1, we check all the test sub-classes against the refactoring pre-conditions defined in Section III-C0c. Cases that satisfy the pre-conditions qualify as refactoring candidates. We report the total number and percentage of cases that qualify as refactoring candidates. If only a small number and portion of refactoring candidates exist, it also implies that our approach is of limited practical value. In particular, in the empirical study, about 25% of test sub-classes are meet the pre-conditions. We aim to show that this percentage in new projects is comparable, if not higher. Otherwise, it indicates that the refactoring opportunities are specific to the empirical study dataset, but not general in new projects.

2) RQ2: Following RQ1, we actually apply the refactoring on the identified candidates from nine representative projects. It is not practical for us to perform the actual refactoring on all Apache projects due to the manual effort for project configuration and test case execution. Here, we report the percentage of refactoring candidates that are actually successfully refactored by our approach. Our automatic refactoring process is constructed based on the features and syntaxes learned from the empirical dataset. Just like any learning process, the empirical dataset can not capture all possible features of an unknown, new dataset. However, we argue that the approach built based on the empirical dataset capture the most general features, and thus should be able to successfully refactor the majority of feasible cases from a new dataset. Otherwise, we cannot claim that our refactoring process can be generally and successfully applied to a new dataset.

3) RQ3: Mutation testing changes specific parts of the source code to ensure a test case will be able to detect the changes, i.e. potential defects. If a test case, before the refactoring, is able to kill a mutant (i.e. detect the change), it should also be able to do so after the refactoring. Otherwise, the test case is not preserving its behavior in terms of identifying potential changes and defects, as represented by mutants in our experiment. Therefore, mutation testing is necessary to evaluate behavior preservation in refactoring. We evaluate test behavior preservation in terms of capturing potential defects (i.e. injected as mutants) in three metrics. First, whether the mutants that are executed by test cases before the refactoring are still executed by test cases after the refactoring. Second, whether the killed mutants before the refactoring are still killed

after the refactoring. Third, whether the survived mutants before the refactoring still survive after the refactoring.

We use PIT [37], a state-of-the-art mutation testing system, to generate mutators. Table IV summarizes all the generated default mutators [38] and their percentage in our dataset.

TABLE IV: Applicable Default Mutators by PIT

Mutator	Description	%
Negate Conditions	mutate all conditions by its logical negation	34%
Void Method Calls	remove method calls to void methods	23%
Empty/Null Returns	replace return values with an 'empty' or 'null'	18%
True/False Returns	mutates a *true* return value to be *false* and vice versa	9%
Math	replaces binary arithmetic operations with another operation	6%
Conditionals Boundary	replaces the relational operators with their boundary counterpart	5%
Primitive Returns	replace int, short, long, char, float and double return values with 0	4%
Increments	replaces increments with decrements and vice versa	1%

Table V shows the mutations in each project. Column 2 shows the number of refactored test cases executing the mutants, ranging from 3 (PDFBox) to 2,510 (Commons). Column 3 shows the number production classes with successfully injected mutants, accounting for 14% to 92% of all production classes called by the test cases in Column 2. The reason why some production classes are not mutated is because they are abstract class or interfaces with little real method implementation and PIT cannot apply mutation operators to them. For example, if there is no method that returns a boolean type, the operator *True/False Returns* (see Table IV) cannot be applied. Column 4 shows that a total of 66,677 mutants—from 468 (Curator) to 35,902 (Commons) in each project—are generated.

TABLE V: Mutation Status Change

Proj.	#Test Cases	#Production Classes(%M)	#Mutations
JackRabbit	431	107 (47%)	9,730
Log4J2	424	152 (35%)	5,023
Qpid-Proton-J	126	43(51%)	1,156
Commons	2,510	700(56%)	35,902
Sakai	45	8 (15%)	501
Curator	44	16 (14%)	468
Avro	27	11 (26%)	2,731
PDFBox	3	12 (92%)	1,358
OpenNLP	173	308 (60%)	9,808
Total	3,783	1,367 (49%)	66,677

4) RQ4: This RQ aims to show how the code base changes quantitatively after the refactoring.

Based on our refactoring rationale, one of the main benefit of the refactoring is to isolate test cases from its dependencies from the production code, as well as eliminating the dependencies within the test code. Thus, we measure four metrics for the decoupling effect of our refactoring solution. First, the amount of inheritance between the test and production code, and how does this increase or decrease after the refactoring. Second, the amount of regular dependencies (i.e. other than inheritance) between the test and production code, and how does this increase or decrease after the refactoring. Third, the amount of inheritance among the test code, and how does this increase or decrease after the refactoring. Fourth, the amount of regular dependencies (i.e. other than inheritance) among the test code, and how does this increase or decrease after the refactoring.

In addition, we expect that the refactoring will impact the size of individual test class, measured by the LOC, #methods, and #fields. Thus, we also measure how these metrics of each test class change comparing before and after refactoring. The goal is to show that our refactoring does not significantly increase the size of the code base, instead it actually reduces the number of unnecessary methods and fields in test classes.

5) RQ5: We report the execution times in identifying refactoring candidates and in performing the refactoring, respectively. Note that the time for identifying refactoring candidates is measured for the entire project; while the time for performing the refactoring is measured for each refactoring candidate.

6) RQ6: User Study Design: Here, we describe the design of the user study involving real-life developers to evaluate the quality and benefits of our refactoring solutions.

Participants: We invite six full-time developers from a software company, who remain anonymous in the study. According to the entrance survey, the participants are well qualified in this study. Four participants have 1 to 4 years working experience as a software engineer/developer. And the other two have 5-9 years and 10+ years of experience. All participants have experience with unit testing. Plus, they all have prior experience with JUnit and Mockito—*four* participants with 1 to 4 years, *one* with 5-9 years, and *one* with 10+ years.

Study Cases: We select a total of six test cases that use test subclasses from our study dataset. These cases are in two distinctive sets, S_1 and S_2, each with three cases. Each set is selected to ensure that the three cases are comprehensive to cover all features illustrated in the problem formalization in Section III-C and all possible refactoring steps introduced in Section IV-A. In addition, the logic of the selected cases are easy to understand such that developers can finish the study in a few hours.

Study Process: Both sets, S_1 and S_2, are given to all six participants. Each participant implements manual refactoring on one set—namely, the *implementation* set, and reviews the auto-refactoring solutions to the other set—namely the *review* set. Of a particular note, the participants are not told that the provided solutions are auto-generated by our tool. S_1 and S_2 serve different roles for different participants. For example, for participant #1, S_1 serves as the *implementation* set and S_2 serves as the *review* set; then for participant #2, S_1 and S_2 switch the roles—S_1 for *review* and S_2 *implement*. We ask each participant to first implement manual refactoring on the *implementation* set and then review the provided solutions for the *review* set. As such, each case is manually refactored by three participants, and the respective auto-refactoring solution is reviewed by another three participants.

The study is held remotely on the AWS servers [39]. The *implementation* cases are loaded and configured in Eclipse. The *review* cases are provided with the *GitHub* links to the original case, as well as the links to the commit id and a diff view of the provided auto-refactoring solution. This is the same environment as the participants normally perform code review in their daily work.

Manual Refactoring Status: As shown in Table VI, for each case, at least one in three participants successfully performs manual refactoring. We obtain a total of 13 successful manual refactoring versions out of the 18 manual refactoring

attempts on the six cases. Among these, *five* manual refactoring versions each takes 5-10 minutes, *one* takes 10-15 minutes, and *seven* takes more than 15 minutes to finish. **In summary, 72% of the manual refactoring cases are successful, and most cases require more than 15 minutes to refactor manually.**

TABLE VI: Successful Manual Refactoring

	Case1	Case2	Case3	Case4	Case5	Case6
# Succ. Participants	2/3	1/3	2/3	2/3	3/3	3/3
Time1 (#Min)	5-10	>15	>15	>15	5-10	5-10
Time2 (#Min)	>15	-	>15	5-10	10-15	5-10
Time3 (#Min)	-	-	-	-	>15	>15

Survey Questions: We ask the participants to take a survey after manual refactoring or reviewing each case. The survey questions are listed below. Note that "(I&R)" indicates the question applies to both the implementation and review cases; "(R)" indicates that the question only applies to the review cases.

- *SQ1 (R): Rate the quality of the provided refactoring solution.(1-6 Scale). And please provide any suggestions (Open Ended).*
- *SQ2 (I&R): Rate your agreement with this statement: Using mocks instead of sub-classing improved the code quality of this example. (1-4 Scale). Explain your rating (Open Ended).*
- *SQ3 (I&R): Rate your agreement with these statements regarding the benefits of mocks: The refactored code—*
  - *makes the test design more cohesive/concise (Scale 1-6).*
  - *makes the test conditions more explicit (Scale 1-6).*
  - *is less coupled from the production code (Scale 1-6).*
- *SQ4 (I&R): Do you see any other benefits or drawbacks from this refactoring?*

SQ1 aims to assess the quality of the auto-refactoring solutions. SQ2, SQ3, and SQ4 evaluate the value and benefits (or drawbacks) of a refactoring solution, regardless of whether it is manual or auto. After reviewing an auto-refactoring solution, the participant is asked to answer all four SQs. While after manual refactoring a case, the participant is asked to answer only SQ2, SQ3, and SQ4, since it is subjective for the developer to rate the quality of his/her own manual refactoring. Thus, each case receives six survey results—three as the manual refactoring and three as the provided auto-refactoring solution—from all six participants. This minimizes individual biases of participants on each case. For SQ2 to SQ4, we investigate the discrepancies between *implement* and *review* cases to quantify the difference between the auto-refactoring solution and the manual refactoring solution on each case.

Finally, we also make a detailed comparison of the manual refactoring solutions and the auto-refactoring solutions to observe what might be the advantages of the manual refactoring solutions. These observations could guide us in potential improvements in the auto-refactoring process.

7) RQ7: Pull Request Study Design: We explain the detailed design of the pull request study.

Study Cases: We select at least one refactoring case from each project in the evaluation dataset. These cases are selected such that they together cover all refactoring features illustrated in the problem formalization in Section III-C, as well as all refactoring steps described in Section IV-A. For the nine open source projects, we select a total of 23 test cases refactored by our approach.

Pull Request Template: For each selected refactoring case, we create a pull request following a consistent template, composed of three parts. First, the overall description of this pull request. Second, the motivation of this pull request, tailored based on the concrete benefits associate with this pull request. Third, the detailed description of the changes made in this pull request. Following is an example from Druid:

Pull Request Example from Druid

Description:
Fixes #11528 (Issue Link). Refactor test class HandlingInputRowIteratorTest.java (Test Sub-class).
Motivation:
1) Decoupling test class from production class.
2) Making test condition more clear by directly verify invocation status of function *handle(InputRow)*.
3) Making test logic more clear by using method stub instead of method overriding.

Key Changes:
1) Created mocking object to replace test subclass
2) Use *Mockito.verify* to verify the invocation status of *handle(InputRow)*, improved test conditions by making it more clear.

Evaluation Metrics: We collect the following metrics in the pull request study to answer RQ6:

1) The response rate of the 23 pull requests and how it compares to the other most recent pull requests in each project. If the response rate of our pull requests is much lower than the average response rate of an open source project, it indicates that our refactoring solutions are not of practical interest to the projects.
2) The turn around time of each pull request, compared with the average turn around time of the other pull requests in a project. If the turn around time of our pull requests is much longer than the average turn around time, it indicates that our refactoring proposal is of low importance in practice.
3) The acceptance rate of the 23 pull requests. If most of our pull requests are rejected after the developers review them, it indicates that our refactoring solutions are not acceptable or are not valuable in practice.
4) The percentage of involved developers who accept our pull requests. For example, to one extreme, if the accepted pull requests are decided by just one single developer, it is not sufficient to prove that the pull requests are well received by a wide range of open source developers.
5) Finally, the detailed comments we receive from the pull requests detailing why the developers accept or reject a pull request. If accept, whether any improvements are required before the pull requests are merged into the open

source projects. If reject, what is the problem with our refactoring solution that leads to its rejection.

VI. Quantitative Evaluation Results (RQ1-RQ4)

A. RQ1: Generality and Applicability

Among the 182 Apache projects that contain test files, 159 (87%) projects contain test sub-classes that inherit from a production class or implements a production interface. This indicates that our approach can be potentially applicable to 87% of Java projects on Apache.

Table VII shows the detailed statistics based on the 159 projects, especially the application of the ten pre-conditions. More specifically, we show the "Total" (row 1), "Min" (row 2), "Max" (row 3), and "Average" (row 4), and "Percentage" (row 5) of the total test classes (column 2), test subclasses (column 3), identified feasible refactoring candidates (column 4), as well as the number of cases excluded by the ten pre-conditions (column 5 to column 14), for the 159 Apache projects.

In these projects, there are a total 86,595 test classes. Among them, 9,932 are test sub-classes that use inheritance for mocking—accounting for 11.5% of test classes. This indicates that inheritance for mocking is not a rare problem; instead, it generally exists in 87% projects, and happens in a non-trivial, 11.5%, population of test classes. More specifically, each project contains, on average 54.6, and up to 1,126, such cases that are potential use cases of our tool.

After checking against the pre-conditions, 2,609 (26%) out of the 9,932 test subclasses qualify as refactoring candidates. In comparison, the percentage of feasible cases that pass the pre-conditions in the empirical dataset is 27%. This indicates that our approach can be generally and consistently applicable to new datasets. More specifically, there are on average 14, and up to 155, refactoring candidates that meet the refactoring pre-conditions in a project. This indicates that our approach has general use cases in real-life projects on Apache.

Summary: In 159 (87% out of the total 182) projects, there exist a total of 9,932 test sub-classes for mocking. This indicates that inheritance for mocking is a general phenomenon observed in Apache projects. In addition, 2,609 (26%) out of the 9,932 test sub-classes are identified as refactoring candidates by our approach. This is consistent with the percentage, 27%, observed based on the empirical dataset. The implication is twofold. Frist, Our refactoring candidate detection built based on the empirical study extends well to new projects, with consistent percentage of feasibility. Second, our approach can be potentially applied to a large number of candidates, a total of 2,609 test subclasses from Apache projects, which indicate its potential practical value.

B. RQ2: Refactoring Successful Rate

Table VIII shows the application of our refactoring framework on the nine projects. As shown on column "#Subcl.", there are totally 774 test sub-classes from these projects. Among these, as shown in columns "#" and "%" under "Candidates", our approach identifies 334 (43%) feasible refactoring candidates that meet all the ten pre-conditions. Out of the 334 feasible refactoring candidates, 276 (83%) are refactored automatically and successfully using our approach, as shown on columns "#,%Succ.".

Columns "#,%Comp." and "#,%Discre." under "Successful Refactoring" show the number and percentage of cases that fail the refactoring due to compile errors and due to inconsistent test case behaviors, respectively. More specifically, 34 (10%) cases failed the refactoring because of syntax issues that were not captured in the empirical dataset, and lead to compile errors. In addition, 24 (7%) test sub-classes, after the refactoring, lead to test behavior discrepancies (column "#,%Discre.") due to special cases. For example, some test cases use the metadata of the test subclass at run-time, and they fail after the refactoring. The reason behind the compile errors and behavior discrepancy is that our approach is built upon the knowledge we gained in the empirical study. Just like any learning process, we cannot guarantee that knowledge extracted based on the learning dataset can 100% cover all situations in an unknown, new dataset. Admittedly, we can keep refining our approach by incorporating the new syntax issues we encounter in the testing dataset, but we leave this to our future work.

Summary: Our approach successfully refactored 83% (276/334) candidates. Note that one can keep refining our approach by incorporating these special cases. However, like any learning process, it is impossible to guarantee 100% generalizability under unknown, new data.

C. RQ3: Test Behavior Preservation with Mutants

Table XII shows the change in the executed, killed, and survived mutants before and after the refactoring.

Column 2 and 3 report the statistics of the mutants being executed before and after the refactoring. Among all 66,677 mutants, 29,019 (44%) are executed before applying our refactoring approach. Note that not all mutants are executed since we use the test cases supplied by the projects, which do not achieve 100% code coverage by test cases. In addition, we only execute test cases affected by the refactoring. The executions of the mutants remains highly consistent after the refactoring, except for 7 (out of totally 66,677) mutants. The discrepancy on these 7 mutants needs further investigation.

Among the executed mutants, we observe (Column 4 — column 7) that, in four projects, including Log4J2, Qpid-Proton-J, Sakai, and Curator, the mutation status (Killed or Survived) remains 100% consistent before and after refactoring. In the other five projects, 0.03% (=3/9,751 in JackRabbit) to 0.7% (=244/35,666 in Commons) mutants changed their status. More specifically, column 4 and 6 show the number of mutants that are killed and survived before the refactoring. Columns 5 and 7 show the number of mutants that change the execution status (killed/survived) after the refactoring. For example, the Commons project, in column 5, *+121* indicates that, after the refactoring, an additional 121 mutants are killed; and *-120* indicates that 120 mutants no longer got killed after the

TABLE VII: Applicability on 159 Apache Projects with Test Subclasses

	#Test Classes	#Test Subclasses	#Candidates	# Cases Excluded By Pre-Conditions									
				P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Total	86,583	9,932	2,609	638	63	210	612	1,781	1,393	1,474	1,076	12	64
Min	1	0	0	0	0	0	0	0	0	0	0	0	0
Max	10,827	1,126	155	131	15	16	287	178	361	467	86	4	8
Avg	475.7	54.6	14.3	3.5	0.3	1.2	3.4	9.8	7.7	8.1	5.9	0.1	0.4
Percentage	-	11.5%	26.2%	8.7%	0.9%	2.9%	8.4%	24.3%	19.0%	20.1%	14.7%	0.2%	0.9%

TABLE VIII: Auto-Refactoring Success Rate in Nine Projects

Proj.	#SubCl.	Candidates		Successful Refactoring					
		#	%	#,%Comp.		#,%Discre.		#,%Succ.	
JackRabbit	67	43	64%	1	2%	1	2%	41	95%
Log4J2	100	33	33%	5	15%	3	9%	25	76%
Qpid-Proton-J	34	9	26%	0	0%	0	0%	9	100%
Commons	405	181	45%	19	10%	14	8%	148	82%
Sakai	51	21	41%	4	19%	1	5%	16	76%
Curator	21	16	76%	3	19%	2	13%	11	69%
Avro	40	8	20%	2	25%	0	0%	6	75%
PDFBox	6	1	17%	0	0%	0	0%	1	100%
OpenNLP	50	22	44%	0	0%	3	14%	19	86%
Total	774	334	43%	34	10%	24	7%	276	83%

TABLE IX: Mutation Status Change

Proj.	Executed		Killed		Survived	
	Before	After	Before	After	Before	After
JackRabbit	3,803 (39%)	✓	2,981	+1, -2	822	+2, -1
Log4J2	1,548 (31%)	✓	992	✓	556	✓
Qpid-Proton-J	332 (29%)	✓	298	✓	34	✓
Commons	17,120 (48%)	+3	12,709	+121, -120	4,411	+123, -121
Sakai	208 (42%)	✓	156	✓	52	✓
Curator	213 (46%)	✓	115	✓	98	✓
Avro	777 (28%)	✓	432	+4, -1	345	+1, -4
PDFBox	688 (51%)	+4	627	+6, -3	61	+4, -3
OpenNLP	4,330 (44%)	✓	2,926	+60, -6	1,404	+6, -60
Total	29,019 (44%)	+7	21,236	+192, -132	7,783	+136, -189

refactoring. The numbers in columns 5 and 7 should sum to the number in column 3—mutants that change their coverage. We sample 30 mutants to investigate the reasons for the change. We find that, in all 30 cases, the behaviors of the tests become non-deterministic after injecting the mutants—the status changes even without refactoring. Thus, the non-determinism is caused by the mutations instead of the refactoring.

Summary: The execution of the mutants before and after applying our approach is highly consistent (< 0.01% of the generated mutants changed their coverage, i.e. if they are executed by the test cases). The test cases, after refactoring, consistently execute, kill, or survive with 99% of the 66,677 mutants injected into the production code. This indicates that our approach generally preserves test behaviors in term of detecting potential defects.

D. RQ4: Code Metrics Change

Overall Code Coupling: In Table X, in column 2 and 4, we show the number of inheritance (column 2) and the number of regular dependencies (column 4) between the test and production code before the refactoring. The percentage numbers in column 3 and column 5 indicate how each type of dependency changes after the refactoring. A negative value indicates the percentage of decreased dependencies. Similarly, the last four columns show the similar information, i.e. the number of inheritance and regular dependencies among the test code before the refactoring (column 6 and column 8), and the changed percentage after the refactoring (column 7 and column 9).

We have the following observations: First, the number of test-to-production inheritance non-trivially decreased by 2% (Avro) to 65% (OpenNLP). Second, the number of test-to-production regular dependencies decreased by up to 53% (OpenNLP). Similarly, in columns 6–9, we examine the number of dependencies among the test classes, and the percentage of increase after the refactoring (column "#(%In.) of T-T Dps."). We separate the inheritance relationship and any other dependencies. We observe that, first, the number of inheritance relationship among test classes decreased in JackRabbit, Commons and Curator, by 8%, 1% and 1%, respectively, and remained stable in the other six projects. Second, the regular dependencies among test classes decreased by 2% (Sakai) to 17% (OpenNLP) in projects except JackRabbit and Avro.

TABLE X: Code Coupling Change

Project	# (%In.) of T-P Dps.				# (%In.) of T-T. Dps.			
	#,%Inherit.		#,%Regular		#,%Inherit.		#,%Regular	
JackRabbit	37	-38%	225	-12%	329	-8%	956	-0%
Log4J2	100	-24%	445	-10%	16	0%	75	-7%
Qpid-Proton-J	34	-26%	264	-9%	22	0%	75	-7%
Commons	399	-35%	4,010	-5%	168	-1%	737	-9%
Sakai	76	-8%	1,018	-1%	70	0%	165	-2%
Curator	20	-45%	1,231	-1%	102	-1%	424	-3%
Avro	267	-2%	3,668	0%	518	0%	1,674	0%
PDFBox	23	-4%	616	0%	12	0%	32	-3%
OpenNLP	17	-65%	36	-53%	2	0%	6	-17%

Individual Test Class Size Change: In Table XI, Columns 2–13 report the average LOC, #Methods, and #Fields before the refactoring, and the average, minimal and maximal percentage of increase after the refactoring over all refactoring cases. A negative percentage indicates that the measure decreases. We observe that after the refactoring, first the LOC averagely increases 1% (PDFBox) to 10% (Avro), except that the LOC of test cases in Sakai averages decreases 4%. This is due to cases where the reusable *mock method* (Section IV-A5) is not applicable. And second, both the #Methods and the #Fields decreases, in average by up to 29% and up to 36% respectively. The decrease is due to Step 1.2 and Step 2.1 (Section IV-A).

Summary: The refactoring overall decreases code complexity. In particular, it non-trivially decouples the test from the production code, by removing up to 65% of the inheritance and up to 53% of the regular dependencies. It also decouples the test code itself—removing the internal inheritance by up to 8% and regular dependencies by up to 17%. Meanwhile, it slightly increases the LOC of the refactored test classes, but more obviously decreases the number of methods and fields.

TABLE XI: Change of Individual Test Class Size Measured by LOC, #Methods, and #Fields

Proj.	LOC				Method				Field			
	#Avg.	Avg. %In.	Min. %In.	Max %In.	#Avg.	Avg. %In.	Min. %In.	Max %In.	#Avg.	Avg. %In.	Min. %In.	Max %In.
JackRabbit	353	4%	-45%	61%	64	-5%	-88%	-1%	6	-9%	-100%	0%
Log4J2	108	8%	-24%	-50%	17	-11%	-78%	0%	4	-20%	-100%	0%
Qpid-Proton-J	178	6%	1%	33%	28	-14%	-80%	0%	6	-22%	-100%	0%
Commons	261	4%	-9%	126%	29	-5%	-67%	12%	7	-11%	-100%	0%
Sakai	128	-4%	-35%	33%	27	-29%	-78%	0%	3	-6%	-50%	0%
Curator	152	8%	1%	24%	13	-11%	-75%	-2%	3	-36%	-100%	0%
Avro	145	10%	2%	39%	24	-20%	-61%	-4%	7	-7%	-100%	0%
PDFBox	309	1%	1%	1%	17	-6%	-6%	-6%	4	0%	0%	0%
OpenNLP	161	3%	-6%	57%	18	-13%	-60%	0%	3	-29%	-100%	0%

E. RQ5: Execution Time

Table XII shows the running time (in seconds) of the framework. Column "#TestCl." shows the total number of test classes in each project—131 (Qpid) to 3,599 (Commons). This is the initial input size to the candidate identification. Among these, we identify 6 (PDFBox) to 405 (Commons) test subclasses, which are processed by the three filtering layers (Section III-C0c). The detection time ranges from 4 to 250 seconds per project. The average execution time of refactoring each case ranges from 0.7 to 1.5 seconds, with the standard deviation of 0 (PDFBox) to 2.2 (Log4J2).

TABLE XII: Auto-refactoring Performance

Proj.	Detection Time (s)			Refactoring Time (s)				
	#TestCl.	#SubCl.	Total-T	#Case	Avg-T	Max-T	Min-T	Std
JackRabbit	1,060	67	42	26	0.7	2.2	0.3	0.6
Log4J2	1,069	100	203	26	1.5	10.6	0.3	2.2
Qpid-Proton-J	131	34	30	9	0.9	1.7	0.4	0.4
Commons	3,599	405	250	139	1.0	11.4	0.2	1.3
Sakai	364	51	44	9	0.5	0.7	0.3	0.1
Curator	201	21	13	9	1.3	4.3	0.4	1.2
Avro	676	40	25	6	1.3	3.8	0.3	1.2
PDFBox	142	6	4	1	0.7	0.7	0.7	0
OpenNLP	277	50	22	18	0.7	4.7	0.2	1
Total	7,519	774	632	243	1.0	11.4	0.2	1.3

Summary: The run-time performance of the framework is a few minutes for detecting all refactoring candidates in a project, and a few seconds for refactoring each case. This suggests that our approach is efficient.

VII. Qualitative Evaluation Results (RQ6 and RQ7)

1) RQ6: User Study Results: **SQ1 (R):** Figure 9 shows the participants' rating on the quality of auto-refactoring solutions. The x-axis is the case ID. The y-axis is the score in the scale of 1 (poor) to 6 (excellent)—overall, a score of 4 or above indicates a positive opinion. The size of (and the number in) each circle shows the three scores given to each case. We observe that, for each case, at least one participant rates positively—score at least 4. In particular, case 4 and case 5 receive unanimous positive scores. And case 3 and case 6 receive 2 (out of 3) positive scores.

We investigate the comments from the participants. We find that the the negative scores fall into three types. First, the criticism is about the original test design, which is irrelevant to the Mockito-based refactoring (Case 1, case 3, and case 6). Second, participants have misunderstandings on the reviewed cases (case 2). There are two sub-classes involved in case 2—one of them cannot be refactored due to *F-1.1* (Section III-C0c). However, the participant thinks that we should also refactor it. The other misunderstanding is that the

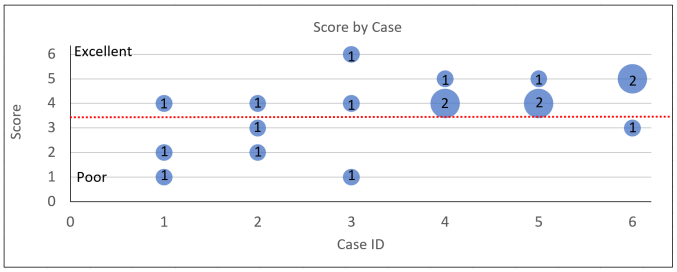


Fig. 9: SQ1: Quality of Auto-Refactoring Solution

participant suggests using *mock method* (Section IV-A5) when it is not appropriate. Third, there exists subjective preferences—one participant favors the separated test subclass and test case before the refactoring in case 1, rather than merging the logic of the subclass with the test case. **Overall, participants rate positively on the refactoring solutions generated by our tool.**

SQ2 (I&R): As shown in Figure 10, in 17 out of the total 18 *implementation* cases, participants agree or strongly agree that using mocks instead of inheritance improves the code quality. The response on the *review* cases is highly consistent—in 13 out of 18 cases, participants agree or strongly agree that the code quality improves. The disagreement on the *review* cases is due to two reasons. First, participants expect to see improvements on the test design/logic itself. Second, participants prefer to separate the mock behaviors in a subclass. **Overall, participants agree that using mocks to replace inheritance improves the code quality.**

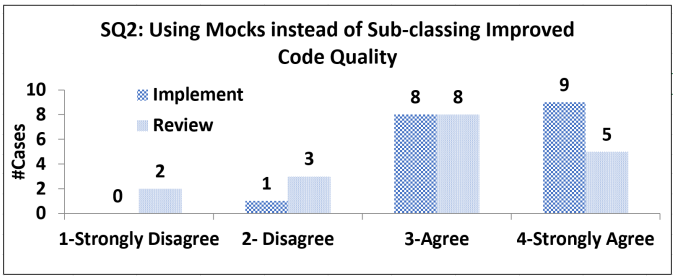


Fig. 10: SQ2: Improved Code Quality (Impl. vs. Review)

SQ3 (I&R): Table XIII summarizes ratings on the refactoring benefits in cohesion/conciseness, explicit, and decoupling. We report the mean rating (column "Mean") of each benefit on the *implementation* (on row 1) and *review* (on row 2) cases separately. In addition, a rating of 4 (*"Somewhat Agree"*) or above indicates positive opinion—thus we also report the percentage of rating of 4 and above (Column "Agree%") on

each benefit. The discrepancies of the ratings between the *implementation* and *review* cases are summarized in row 3.

TABLE XIII: Refactoring Benefits (Implementation vs. Review)

Benefits	Cohesion/Concise		Explicit		Decoupled	
	Mean	Agree%	Mean	Agree%	Mean	Agree%
Implementation	5.3	100%	5.2	85%	4.5	77%
Review	3.8	61%	3.8	67%	4.1	72%
Discrepancy	1.5	39%	1.5	18%	0.4	5%

We observe that for the majority cases—at least 77% and 61% of the *implementation* and *review* sets respectively—participants agree with these benefits. However, the agreement is weaker on the *review* cases compared to the *implementation* cases. The *review* cases receive 0.4 to 1.5 lower mean rating, and 5% to 39% less positive rating. This indicates that the manual refactoring by developers boosts these benefits on more cases and to a higher degree. **In conclusion, our auto-refactoring framework alone helps developers reap the three benefits to some extent. For further benefits, developers may manually improve the test itself.**

SQ4 (I&R): The participants report additional benefits for the *implementation* cases, including improved readability and understandability (four cases), maintainability (one case), and test power (two cases). Similarly, in *two*, *one*, and *one review* cases, participants report these three benefits as well. The readability/understandability and maintainability are associated with the three benefit aspects surveyed in SQ4. The improved test power is because Mockito allows additional verification of the mock object execution/status. A notable drawback on the review cases is that the original code comments, which explain the test logic, are not preserved after the refactoring. **In summary, our approach can improve the readability/understandability and maintainability the test code, and can make the test more powerful. However, the drawback is that the original code comments cannot be retained.**

Comparison of manual refactoring solutions and auto-refactoring solutions: We find that the manual refactoring solutions often contain improvements to the test logic and design, such as simplifying the test logic and removing redundancy. These improvements are not related to how mocking frameworks are used. In addition, the manual refactoring solutions also contain more sophisticated usage of certain mocking APIs, such as verifying the execution order of mock objects, using argument captor, and checking input argument value and type, which are absent from our auto-refactoring solutions. However, such usage requires in-depth understanding of the test cases and their dynamic execution status. These observations help us to understand the rating discrepancy observed for SQ2 and SQ3.

Potential Improvements: The comparison leads to a summary of (potential) improvements. First, we should import Mockito static methods in refactoring to simplify the code. At the time of writing, this issue is fixed. Second, we can further enhance the usage of Mockito, such as verifying execution order of mock objects, using argument captor, checking input argument value and type. However, this potentially relies on dynamic analysis. Finally, more future work should focus on improving the general test design, such as removing redundant code and simplifying the test logic. We plan to address the last two directions in our future work.

Summary: 1) Participants generally rate positively on the refactoring solutions generated by our framework. 2) Participants agree that the refactoring solutions generated by the framework improve the cohesion/conciseness of test code, make test condition more explicit, and decouple test code from production code. They also point out additional benefits, including readability/understandability and more powerful test. However, to further enhance these benefits, developers need to improve the test logic itself with manual effort. Thus, our tool can serve as an efficient first step in refactoring. 3) An obvious drawback of our framework is that the original code comments cannot be retained after refactoring.

2) RQ7: Pull Request Results: Table XIV shows the overall status of the 23 pull requests in five categorizes:

1) "accepted"—the pull request was accepted by the project;
2) "rejected"—the pull request was rejected;
3) 'undecided"—there exists conflicting opinions among the reviewers, such that whether to accept the pull request is undecided;
4) "exception"—the pull request was closed unexpectedly;
5) "pending"—the pull request did not receive any response as the time of this writing.

For "exception", the developers decide to completely remove the test-subclasses being refactored, since the test-subclasses are found to be unnecessary after reviewing our pull requests. And this decision is irrelevant to our refactoring solution.

Next, we analyze the results in Table XIV using the five metrics described in Section V-C.

TABLE XIV: Pull Requests Status

Result	Project	Pull Request	#Comments	
			Accept	Reject
Accepted 9 (39%)	Accumulo	Accumulo-2254	3	-
	Cloudstack	Cloudstack-5480	3	-
	Curator	Curator-397	1	-
	Druid	Druid-11529	1	-
	Druid	Druid-11561	1	-
	Dubbo	Dubbo-8446	2	-
	Log4J	Log4J2-584	1	-
	OpenNLP	OpenNLP-396	1	-
	Sakai	Sakai-9832	1	-
Rejected 6 (26%)	CMs-Codec	Codec-94	-	1
	CMs-JXPath	JXPath-22	-	1
	CMs-Pool	Pool-98	-	2
	CMs-VFS	VFS-213	-	1
	PDFBox	PDFBox-130	-	1
	Proton-J	ProtonJ-41	-	1
Undecided 1 (4%)	CMs-BeanUtils	BeanUtils-96	1	1
Exception 2 (10%)	Avro	Avro-1341	-	-
	CMs-RNG	RNG-102	-	-
Pending 5 (22%)	Cayenne	Cayenne-462	-	-
	CMs-Collections	Collections-249	-	-
	CMs-Configuration	Configuration-142	-	-
	CMs-Lang	Lang-801	-	-
	JackRabbit	JackRabbit-95	-	-
Total			15 (65%)	8 (35%)

Response Rate: Among all 23 pull requests, 5 have not received any responses yet—thus the response rate of our pull

TABLE XV: Pull Requests Response Turn Around Time

Project	Rec. PRs' TA Time	MR PRs' TA Time	Diff TA Time
Sakai	1.3	0.5	-0.8
Curator	5.1	0.5	-4.6
Avro	9.5	0.1	-9.4
PDFBox	2.1	0.5	-1.6
OpenNLP	11.8	0.0	-11.8
Dubbo	0.1	0.1	0
Druid	1.8	32	30.2
Accumulo	3.6	0.4	-3.2
Cloudstack	0.2	0.2	0
Log4J2	9.5	0.5	-9.0
ProtonJ	3.4	0.4	-3.0
Commons	1.9	0.2	-1.7

TABLE XVI: Received Feedback and Suggestions in Pull Requests

Feedback Type	Feedback	#Developers
Positive Feedback	Looks good to me	4
	Using Mockito is big improvement	2
	Refactoring is a good clean up	1
	Using Mockito allows backward-compatible evolution	1
	Using Mockito make test logic clear	1
Negative Feedback	Mockito raises the bar of contribution	4
	Use Mockito makes code hard to understand	2
Side Benefits	Improve test performance/flaky test	1
Future Improvement Opportunities	Translate Mocking frameworks	2
	Add Javadoc	1
	Empower test by leveraging Mockito	1

requests is 78%. In comparison, we collected the most recent 59 pull requests (we only collect 59 requests due to the GitAPI limit) from each project—resulting a total of 786 recent pull requests in total. We found that only 448 of these pull requests have responses—indicating a response rate of 57% for all pull requests. The results suggest that *our pull requests receive a higher response rate (21%) than the average.*

Turn Around Time: For the pull requests with responses, we report the average turn around time between creating the request and getting the first response in Table XV. Column 2 reports the average turn around time (in days) of the recent pull requests. Column 3 shows the average turn around time of our pull requests. Column 4 shows the difference between column 3 and column 2—a negative value indicates how much sooner our request receive response compared the other requests. We observe that our pull requests get faster response in all projects, except for Druid. We made two pull requests to Druid, one got response within a day and the other was delayed for 64 days. The delay of 64 days is due to extraneous reasons that, we believe, are irrelevant to our pull request itself. This indicates that our mocking refactoring pull requests draw maintainers' more immediate attention comparing to most of the other common pull requests.

Acceptance Rate of Pull Requests: As shown in the first column of Table XIV, 9 (50%) requests were accepted and merged into the open source projects, 6 (33%) were rejected, 1 (6%) were undecided, and 2 (11%) had exceptions.

Acceptance Rate by Developers: The last two columns of Table XIV shows the number of comments left by the project developers on accepting or rejecting each pull request. For example, for pull request *Accumulo-2254*, we received three different comments on accepting it. Overall, for the 17 pull requests with responses, we received 23 comments in total, and 15 (65%) comments were accepting and 8 (35%) comments were rejecting the pull requests. Of a particular note, these comments are from 21 different developers. The 8 rejecting comments are from only 4 (21%) developers; while the 15 accepting comments are from the other 15 (79%) developers.

Analysis of Comments: Table XVI summarizes the different types of comments we received from the developers who reviewed our pull requests. We categorize them into four types:

1) Positive comments: Overall, 4 developers think that our refactoring solutions "LGTM" (looks good to me). Two developers think that using mocks instead of real classes in test is "a big improvement". One developer underscores that mocking is a better approach since it allows backward-compatible evolution without breaking tests. The same developer also points out that mocking by a framework makes test intent clearer.

2) Negative comments: There are two concerns. First, Using inheritance is convenient and easy to understand—in comparison, Mockito makes code hard to understand. Second, Using a mocking framework raises the bar of contribution and thus makes code harder to maintain. However, these comments target at the concerns of using mocking and Mockito in general, but do not point to any specific problems of our refactoring solution. Note that, 4 of the 6 rejected pull requests in Table XIV are handled by the same developer.

3) Side Benefits: One developer, after reviewing our refactoring proposal, identified an additional improvement on the test logic itself to test the performance and address test flakiness. We believe that this is a side benefit of using Mockito to make the test logic more clear and understandable.

4) Future Improvement Opportunities: We received some suggestions in merging the pull requests. These suggestions point to some future improvement opportunities in our auto refactoring framework, including:

- Translating the refactoring solutions to different mocking frameworks, such as from Mockito to EasyMock depending on individual projects' preferences. (Druid-11529, Accumulo-2254)
- Adding comments and Javadoc for better documenting the refactored test cases. Of a particular note, we got a similar comment in our user study in RQ5—one developer pointed out that a drawback of our framework is that the original code comments are removed after refactoring. (OpenNLP-396)
- Empowering test cases by leveraging advanced Mockito functions, such as *verifyNoMoreInteractions* for checking whether all executed methods of the mocking object have been verified. However, such use cases highly depend on the detail of each test case. Therefore, it is not practical to make the usage of advanced Mockito functions fully automated. We envision that developers could benefit from an interactive refactoring tool to provide related

suggestions for developers and allow them to make interactive refactoring decisions. (Log4J2-584)

Summary: The response rate of our pull requests is 70%, which is 13% higher than the other common pull requests. The average turn around time for our pull requests is shorter than the common pull requests except Druid, which indicates that the mock refactoring pull requests are very attractive. Within the responded pull requests, 9 (60%) were merged and 6 (40%) were rejected, with 15 (65%) approval and 8 (35%) rejection votes. The positive feedback from the approval comments conclude that our refactoring solutions are in good quality, make test logic clear, and allow backward-compatible evolution. The negative feedback from the rejection comments is concerned that using mocking or mockito makes the code hard to understand and raises the bar of contribution as it requires the knowledge of mocking frameworks. The comments of pull requests also point out the potential improvement opportunities, including supporting refactoring for multiple mocking frameworks, adding Javadoc for better documentation, and making the refactoring interactive to empower the tests by leveraging functions of mocking frameworks.

VIII. LIMITATIONS, THREATS TO VALIDITY, AND FUTURE DIRECTIONS

We acknowledge that our framework has several limitations. First, it only focuses on replacing inheritance by using Mockito for mocking. It does not improve the test case logic/design itself. In addition, compared to manual refactoring created by developers, our framework is limited in leveraging the advanced features of Mockito, such as verifying execution order of mock objects. In manual refactoring, the usage of advanced features is based on manual understanding of the test intention. This could potentially be automated by dynamically analyzing the test case execution. However, our framework currently is purely based on static code analysis. Third, our framework won't preserve the code comments after the refactoring. Fourth, although we generated a total of 66,677 mutants to evaluate the test behaviour preservation of test cases, we cannot guarantee that the test behaviour preserves under all possible defects. Since it is impossible to exhaustively evaluate the test behaviors under all possible defects through generating mutants. Lastly, our framework is limited to Java and Mockito. However, the overall design principle of mocking and refactoring rationale in this work should still hold for other languages and mocking frameworks. We plan to address these limitations in the future.

Only very limited empirical evidence is available to show that inheritance-based mocking leads to code that is more difficult to maintain than using a mocking framework [10]. In this work, we investigate this problem in a qualitative study involving real developers (RQ6) and the results indicate that using a mocking framework can improve test code quality and achieve the three aspects of benefits that relate to maintainability (SQ2 and SQ3 in Section 6.4.1) compared to using inheritance. However, we acknowledge that the conclusion may vary depending on the group of participants. Another external threat to validity is that the benefits of using a mocking framework over inheritance requires that the user has preliminary understanding of the mocking framework. If a user has zero prior knowledge, he/she may find inheritance easier to use and understand. Particularly, the participants in the qualitative evaluation all have prior experience with Mockito. This poses an internal threat to validity towards the findings reported in Section VII.

We also would like to acknowledge that the pre-conditions and the refactoring syntax in our approach are derived based on the 208 cases from the empirical study. We cannot guarantee that the empirical study captures all possible detailed syntax variations for the pre-condition and refactoring process. For example, we initially have 10% of the feasible refactoring candidates failed the refactoring due to syntax that were not observed in the empirical study dataset. Some typical syntax errors are missing imports when test subclass and the test case that uses it are defined in different packages, or naming duplication after refactoring when there are multiple test subclass instances created as fields in a test class. However, the key point, as elaborated in RQ1, is that our approach based on the initial empirical study is generally applicable to new datasets. Just like in any learning process, we can make the pre-condition details and refactoring process more inclusive in capturing more possible variations by continuing learning based on more new datasets. We fixed 5 (13%) out of the 39 syntax errors related to import syntax and leave the rest to the future work.

Our approach only identifies 26% of total test sub-classes as feasible refactoring candidates, which is not a high percentage, after applying the ten refactoring pre-conditions on the 159 Apache projects. The relatively low feasible rate could be constrained by our knowledge in building the approach and also by the limitations of Mockito. It is possible that some of the infeasible cases could be addressed with future research that either advances the refactoring methodologies or addresses limitations of existing mocking frameworks. Thus, we would like to point to two future research directions:

- Refactoring test subclasses with advanced design features. For example, our pre-conditions "P3", "P8", "P9", and "P10" point to test subclasses that with additional behavior (i.e. new methods or new attributes) than the mocked objects, batch instantiation, and inner class definition. These features imply complicated mocking behaviors that pose challenges to the refactoring procedure proposed in this work. However, they may still be feasible for refactoring with more advanced analysis of the test cases and design of the refactoring solutions. For example, a refactoring solution with good design may require beyond dynamic analysis of the test behavior and the usage of advance mocking framework APIs, such as variable capture. We leave this to future research.
- Addressing the constraints of existing mocking frameworks that limit the feasible refactoring. For example, "P1" (mock multiple objects), "P4" (mock with self-

reference), "P5" (mock object creation with dynamic binding), "P6" (mock with special annotations), and "P7" (mock with access to protected fields) identify cases that cannot or should not be refactored due to the limitations of existing mocking frameworks. For example, how to allow Mockito to support special annotation could be an interesting research direction. It calls for more future research, and more thorough investigation of the feasibility and challenges to over come the limitations with existing mocking frameworks to enable related refactoring.

IX. Related Work

Software Refactoring: Significant software development cost is devoted to software maintenance [40]–[43], as software becomes more complex and drifts away from its original design [44]–[47]. Refactoring is an important maintenance activity that restructures a system and improves code quality [46], [48], [48]–[54]. Kim showed that refactoring is challenging and there generally lacks tool support [55]. In past years, researchers proposed methods and tools to automate the refactoring process [56]–[67]. Most refactoring tools focus on detecting and refactoring God Classes [51], [61], [68]–[70], and eliminating Code Clone [57]–[59], [71]–[73]. Tsantalis et. al proposed a refactoring approach to replace state checking (i.e. *if/else*) with polymorphism to reduce code complexity [62]. **Despite numerous prior works, we are the first to focusing on refactoring the usage of inheritance by using mocking framework, to improve unit testing design.**

Test Code Smells: Code smell is a surface indication that usually corresponds to a deeper problem in the system [51]. Test smells are the sub-optimal design choices in test code [74], [75]. They can make test cases less effective and more difficult to understand [74], [76]–[81]. There are various techniques and tools to support automated test smell analysis [74], [82]–[88]. Van Deursen et. al defined a catalog of 11 test smells [74]. Based on this catalog, Van Rompaey et. al introduced a metric-based technique to identify two smells, *General Fixture* and *Eager Test* [84]. Greiler et. al developed a Maven plugin to detect test fixture related smells and provide guidance for refactoring them [85]. Santana et. al implemented an Eclipse plugin to refactor *Assertion Roulette* and *Duplicate Assert* [87]. Other works focus on analyzing the impact of test smells [76], [89]–[91]. **To our best knowledge, no prior work investigated sub-optimal practice in unit test mocking.**

X. Conclusion

We proposed a refactoring framework and implemented it as an Eclipse-Plugin [92] to automatically search for the usage of inheritance and replace it by Mockito for mocking. The framework is built upon the empirical experience drawn from *five* open-source projects. We evaluated our framework on *nine* open-source projects, both quantitatively and qualitatively. The quantitative evaluation proved that our framework was generally applicable to new dataset that was independent from the empirical study. The refactoring solution generally preserves test behaviors in term of detecting defects (in terms of mutants). The refactoring reduced the code complexity—particularly decoupled test code from production code. Lastly, the framework provided efficient run-time performance on real-life projects.

The qualitative evaluation, involving experienced developers, suggested that auto-refactoring solutions by our framework were of good quality. Furthermore, the refactoring solutions improved the quality of the unit test cases in various aspects, such as improving the cohesion/conciseness, readability/understandability and maintainability of the test code, made test condition more explicit and the test cases more powerful.

The study based on 23 pull requests to the fourteen open source projects (including the five projects for the empirical study and the nine projects for evaluation) showed that the refactoring solutions generated by our approach are quite well received in the open source projects and by their developers. Among the 23 requests, 9 requests were accepted/merged, 6 requests were rejected, the remaining requests were pending (5 requests), with unexpected exceptions (2 requests), or still undecided (1 request). In particular, among the 21 open source developers that were involved in the reviewing process, 81% gave positive votes. The detailed comments from the developers underscores the strength of our refactoring solutions, such as enforcing back-ward compatibility when the test/production code changes. We also identified potential future improvement directions, including enabling the translation between different mocking frameworks; adding comments and Javadoc for the refactored test cases, and empowering the refactored test cases with advanced mocking functions.

Data Availability

Our plugin is open sourced under the MIT license [7]. The code base of the plugin is hosted on GitHub at https://github.com/wx930910/JMocker. In addition, the version of our plugin used in this study, and all the related data for this study is available in Zenodo at https://doi.org/10.5281/zenodo.7349416.

Acknowledgments

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

References

[1] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.
[2] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.
[3] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 402–412. IEEE, 2017.
[4] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. Mock objects for testing java systems. *Empirical Software Engineering*, 24(3):1461–1498, 2019.
[5] https://easymock.org/.
[6] https://site.mockito.org/.
[7] https://powermock.github.io/.

[7] https://en.wikipedia.org/wiki/MIT_License

[8] https://junit.org/junit5/.
[9] https://wiki.python.org/moin/PyUnit.
[10] Gustavo Pereira and Andre Hora. Assessing mock classes: An empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 453–463. IEEE, 2020.
[11] Jagadeesh Nandigam, Venkat N Gudivada, Abdelwahab Hamou-Lhadj, and Yonglei Tao. Interface-based object-oriented design with mock objects. In *2009 Sixth International Conference on Information Technology: New Generations*, pages 713–718. IEEE, 2009.
[12] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, not objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246, 2004.
[13] Madhuri R Marri, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. An empirical study of testing file-system-dependent software with mock objects. In *2009 ICSE Workshop on Automation of Software Test*, pages 149–153. IEEE, 2009.
[14] Shaikh Mostafa and Xiaoyin Wang. An empirical study on the usage of mocking frameworks in software testing. In *2014 14th international conference on quality software*, pages 127–132. IEEE, 2014.
[15] Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, and Sunny Wong. An automatic refactoring framework for replacing test-production inheritance by mocking mechanism. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 540–552, 2021.
[16] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.
[17] Jeff Grigg. http://wiki.c2.com/?ArrangeActAssert/, 2012.
[18] https://dubbo.apache.org/.
[19] https://druid.apache.org/.
[20] https://accumulo.apache.org/.
[21] https://cayenne.apache.org/.
[22] https://cloudstack.apache.org/.
[23] https://javadoc.io/static/org.mockito/mockito-core/3.1.0/org/mockito/MockSettings.html#extraInterfaces-java.lang.Class...-.
[24] https://github.com/mockito/mockito/wiki/FAQ#what-are-the-limitations-of-mockito.
[25] https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations.
[26] https://www.javadoc.io/doc/org.powermock/powermock-reflect/1.6.5/org/powermock/reflect/Whitebox.html#getInternalState-java.lang.Object-java.lang.Class-.
[27] https://projects.eclipse.org/projects/eclipse.jdt.
[28] https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/stubbing/OngoingStubbing.html#thenAnswer-org.mockito.stubbing.Answer-.
[29] Java Code Geeks. Mockito programming cookbook. https://www.javacodegeeks.com/wp-content/uploads/2016/09/Mockito-Programming-Cookbook.pdf.
[30] https://jackrabbit.apache.org/jcr/index.html.
[31] https://logging.apache.org/log4j/.
[32] https://qpid.apache.org/.
[33] https://www.sakailms.org/.
[34] http://curator.apache.org/.
[35] https://avro.apache.org/.
[36] https://pdfbox.apache.org/.
[37] https://pitest.org/.
[38] https://pitest.org/quickstart/mutators/.
[39] https://aws.amazon.com/lightsail/.
[40] Clemente Izurieta and James M Bieman. How software designs decay: A pilot study of pattern evolution. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 449–451. IEEE, 2007.
[41] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE transactions on software engineering*, 25(4):493–509, 1999.
[42] Qiang Tu et al. Evolution in open source software: A case study. In *Proceedings 2000 International Conference on Software Maintenance*, pages 131–142. IEEE, 2000.
[43] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22. IEEE, 2005.
[44] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.
[45] Tor Guimaraes. Managing application program maintenance expenditures. *Communications of the ACM*, 26(10):739–746, 1983.
[46] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.
[47] Gábor Szoke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Designing and developing automated refactoring transformations: An experience report. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 693–697. IEEE, 2016.
[48] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, 2020.
[49] William F Opdyke. Refactoring object-oriented frameworks. 1992.
[50] William C Wake. *Refactoring workbook*. Addison-Wesley Professional, 2004.
[51] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
[52] Karim O Elish and Mohammad Alshayeb. Investigating the effect of refactoring on software testing effort. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 29–34. IEEE, 2009.
[53] Frens Vonken and Andy Zaidman. Refactoring with unit testing: A match made in heaven? In *2012 19th Working Conference on Reverse Engineering*, pages 29–38. IEEE, 2012.
[54] Mesfin Abebe and Cheol-Jung Yoo. Trends, opportunities and challenges of software refactoring: A systematic literature review. *international Journal of software engineering and its Applications*, 8(6):299–318, 2014.
[55] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.
[56] Yoshio Kataoka, Michael D Ernst, William G Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 736–743. IEEE, 2001.
[57] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 98–107. IEEE, 2000.
[58] Robert Tairas and Jeff Gray. Get to know your clones with cedar. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 817–818, 2009.
[59] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307, 2012.
[60] George Ganea, Ioana Verebi, and Radu Marinescu. Continuous quality assessment with incode. *Science of Computer Programming*, 134:19–36, 2017.
[61] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.
[62] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of refactoring opportunities introducing polymorphism. *Journal of Systems and Software*, 83(3):391–404, 2010.
[63] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.
[64] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):1–33, 2014.
[65] Liming Zhao and Jane Huffman Hayes. Rank-based refactoring decision support: two studies. *Innovations in Systems and Software Engineering*, 7(3):171–189, 2011.
[66] Philip Mayer and Andreas Schroeder. Automated multi-language artifact binding and rename refactoring between java and dsls used by java frameworks. In *European Conference on Object-Oriented Programming*, pages 437–462. Springer, 2014.

[67] Marcelo Serrano Zanetti, Claudio Juan Tessone, Ingo Scholtes, and Frank Schweitzer. Automated software remodularization based on move refactoring: a complex systems approach. In *Proceedings of the 13th international conference on Modularity*, pages 73–84, 2014.

[68] Alexander Chatzigeorgiou, Spiros Xanthos, and George Stephanides. Evaluating object-oriented designs with link analysis. In *Proceedings. 26th International Conference on Software Engineering*, pages 656–665. IEEE, 2004.

[69] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. A two-step technique for extract class refactoring. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 151–154, 2010.

[70] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Playing with refactoring: Identifying extract class opportunities through game theory. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5. IEEE, 2010.

[71] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. Jdeodorant: clone refactoring. In *Proceedings of the 38th international conference on software engineering companion*, pages 613–616, 2016.

[72] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 53–62. IEEE, 2012.

[73] Sandro Schulze and Martin Kuhlemann. Advanced analysis for code clone removal. In *Proceedings des Workshops der GI-Fachgruppe Software Reengineering (SRE), erschienen in den GI Softwaretechnik-Trends 29 (2)*, pages 10–12. Citeseer, 2009.

[74] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, 2001.

[75] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[76] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, 2018.

[77] Stefan Berner, Roland Weber, and Rudolf K Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579, 2005.

[78] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[79] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, 2017.

[80] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190, 2015.

[81] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[82] Vahid Garousi, Baris Kucuk, and Michael Felderer. What we know about smells in software test code. *IEEE Software*, 36(3):61–73, 2018.

[83] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[84] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.

[85] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331. IEEE, 2013.

[86] Negar Koochakzadeh and Vahid Garousi. A tester-assisted methodology for test redundancy detection. *Advances in Software Engineering*, 2010, 2010.

[87] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. Raide: a tool for assertion roulette and duplicate assert identification and refactoring. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 374–379, 2020.

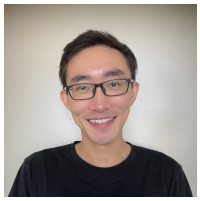
[88] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *J. Object Technol.*, 6(9):231–251, 2007.

[89] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15, 2016.

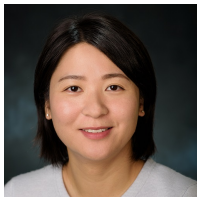
[90] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[91] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pages 5–14. IEEE, 2016.

[92] https://github.com/wx930910/JMocker.

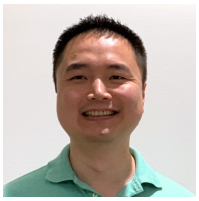


Xiao Wang is a Senior Software Development Engineer at Amazon. He received his Ph.D. in System Engineering with a concentration on Software Engineering from Stevens Institute of Technology in 2022, advised by Dr. Lu Xiao. His research interests lie in software architecture, software refactoring, software testing and cyber-physical systems. He published his work in different journals and conferences, including TSE, ICSE, FES and ICSA.



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, advised by Dr. Yuanfang Cai. She received the first-place prize at the ACM Student Research Competition in 2015. She earned her Bachelor's degree in Computer Science from Beijing University of Posts and Telecommunications in 2009.

Tingting Yu Tingting Yu (Member, IEEE) received the Ph.D. degree in computer science from the University of Nebraska-Lincoln. She is an associate professor of computer science at University of Cincinnati. Her research focuses on software engineering, with a focus on developing methods and tools for improving the reliability and security of complex software systems, testing for concurrent software, regression testing, and performance testing.



Sunny Wong is a Senior Software Architect at Envestnet, with over a decade of experience in the aerospace/defense, healthcare, and finance industries. He was named Young Engineer of the Year in 2019 by the IEEE Philadelphia Section. Sunny received his Ph.D. in computer science from Drexel University. His research interests include software architecture and design modeling, tools support for improving developer productivity, and applications of AI techniques to software development.

Anne Woepse manages the Enterprise Products team at AGI. She previously worked as a software engineer for 7 years on enterprise software solutions at AGI. She received her Bachelors in Mathematics and Physics from Bryn Mawr College, and her Master's from the University of Pennsylvania in Computer and Information Technology. She has performed prior research on software development challenges with air-gap isolation.