An Automatic Refactoring Framework for Replacing Test-Production Inheritance by Mocking Mechanism

Xiao Wang
xwang97@stevens.edu
Stevens Institute of Technology
Hoboken, NJ, USA

Lu Xiao
lxiao6@stevens.edu
Stevens Institute of Technology
Hoboken, NJ, USA

Tingting Yu
tingting.yu@uc.edu
University of Cincinnati
Cincinnati, OH, USA

Anne Woepse
anne.woepse@ansys.com
Analytical Graphics, Inc.
Exton, PA, USA

Sunny Wong
sunny@computer.org
Analytical Graphics, Inc.
Exton, PA, USA

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 "faked" object. Despite the existence of powerful, dedicated mocking frameworks, developers often turn to a "hand-rolled" approach. 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 *four* other projects. Evaluation results show that our framework is efficient, generally applicable to new datasets, and generally preserves test behaviors in term of detecting defects (in the form of mutants). In addition, the refactoring decouples test code from production code. The qualitative evaluation with experienced developers suggests that the auto-refactoring solutions generated by our framework are of good quality. The refactoring solutions improve the quality of the unit test cases in various aspects, such as cohesion/concise, readability/understandability and maintainability, as well as making test conditions more explicit and test cases more powerful.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**; *Software evolution.*

KEYWORDS

software refactoring, software testing

ACM Reference Format:
Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, and Sunny Wong. 2021. An Automatic Refactoring Framework for Replacing Test-Production Inheritance by Mocking Mechanism. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468590

1 INTRODUCTION

Software testing is a critical element of software quality assurance [1, 2]. Unit testing is an important type of testing that focuses on individual units of a software system [3]. A unique challenge to unit testing is that software elements are inter-dependent on each other [3, 4]. 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 [5, 6], i.e., replacing the dependency by a "faked" object. For example, instead of waiting until the external database is deployed, developers create a "faked" 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* [7–9], 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 in orthogonally with classic automated unit testing frameworks, such as JUnit [10] and PyUnit [11].

Despite the existence of powerful mocking frameworks, developers often turn to a "hand-rolled" approach—inheritance. 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 *nine* open source projects examined in our study (Section 3.1, 6.1), which already adopted a mocking framework, according to our study, about half of the cases when mocking is needed, developers still use inheritance as a hand-rolling approach instead of using a mocking framework. However, inheritance is not intended for mocking and often requires tedious implementation when being used for this purpose. As a result, it may compromise the design quality of unit tests. As illustrated in Section 2, inheritance has the following drawbacks compared to using a mocking framework such as Mockito: 1) Implicit test condition and blurred test logic; 2) Difficult-to-maintain test code that couples with the production code; and 3) Incohesive test design that separates the mocking behavior from the test case that leverages it.

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 [12]. It is adopted in both commercial and open source projects [5]. 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 3) 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 4). Next, we propose a fully automated refactoring framework and implement it as an Eclipse-Plugin (Section 5). 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 6) of the proposed framework using another *four* open-source projects. The quantitative evaluation shows the general applicability, high correctness, and efficient run-time performance of the refactoring framework. The qualitative evaluation—participated by experienced, full-time developers—proves that the auto-refactoring solutions generated by our approach are of good design quality and provide various benefits for improving test code design.

In summary, this work makes the following contributions: ① An empirical study involving *five* open-source projects investigating whether it is feasible and how to automatically replace inheritance by Mockito for mocking (Section 3). ② A formalization of the problem definition of the auto-refactoring to replace inheritance by using Mockito (Section 4). ③ A fully automated refactoring framework and its Eclipse-Plugin implementation to identify feasible refactoring candidates and perform the refactoring on each candidate (Section 5). ④ Quantitative and qualitative evaluation of the proposed framework on *four* open-source projects (Section 6).

2 BACKGROUND AND MOTIVATION

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

2.1 Unit Testing

Unit testing aims at validating that each unit of function performs as expected [3, 13]. 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 [14]. 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.

Motivating Example: In an e-Commerce system, *CustomerService* defines a service, *subscribeCustomer*, to subscribe customers by email. This service 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 *EmailManager* is not fully implemented yet—neither the database nor the external service is available. Thus, we isolate the *FUT*, *subscribeCustomer*, from *EmailManager* by mocking the latter. Next, we illustrate mocking through inheritance and Mockito:

2.2 Mocking by Inheritance

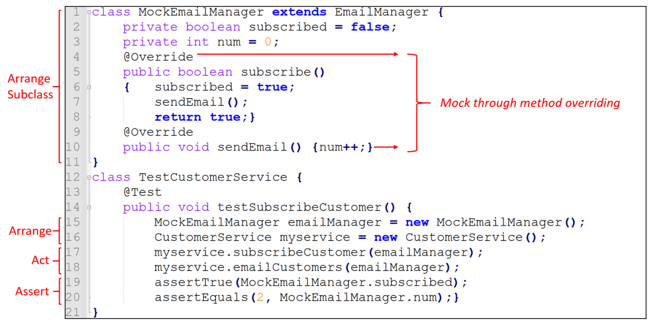
Inheritance is a mechanism to derive a subclass from a base class. The subclass inherits the attributes and methods of the base class. Meanwhile, method overriding allows the subclass to replace certain method implementation of the base class. Inheritance is used as a "hand-rolled" approach for mocking. 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).

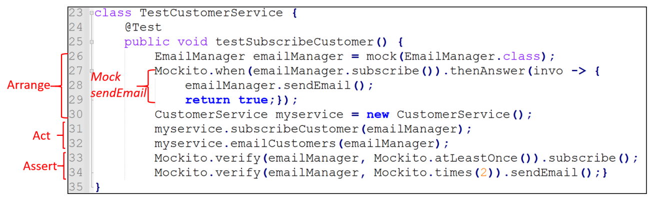
2.3 Mocking by Mockito

Mockito offers three aspects of capabilities for mocking. ① Mockito allows easy creation of a mock object as a "mock" or a "spy". The "mock" is a completely faked object and is entirely instrumented to track the interactions with it. In comparison, the "spy" wraps a real



```
 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 }
```

(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 }
```

(b) Mocking by Mockito

Figure 1: A Motivating Example

instance of the mocked object. The "spy" should be used when the execution of real methods is necessary in testing. ② 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. ③ 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..

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*.

2.4 Benefits of Mockito Over Inheritance

Mockito has the following benefits over inheritance for mocking:

① 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.

② 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.

③ 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.

3 EMPIRICAL STUDY

The goal is to gain empirical experience regarding whether it is feasible and how to automatically replace inheritance by Mockito.

3.1 Dataset and Study Purpose

We select *five* open source projects as our empirical study subjects—they are Dubbo [15], Druid [16], Accumulo [17], Cayenne [18], and CloudStack [19]. 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*.

For each case where a test subclass inherits/implements a production class/interface for testing, we investigate a key question: *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?*

In the empirical study, one author, the driver, manually reviews and refactors each test subclass.

(1) If the refactoring is not feasible or not successful, the driver records detailed reasons.
(2) The driver summarizes the key refactoring steps of each refactored case, and determines whether it can be automated or not. If not, the driver records the reasons.
(3) The team meets weekly to inspect and discuss the refactoring solutions obtained from step 1 and 2 to: i) improve the manual refactoring solutions, and ii) define and discuss the auto-refactoring formalization.

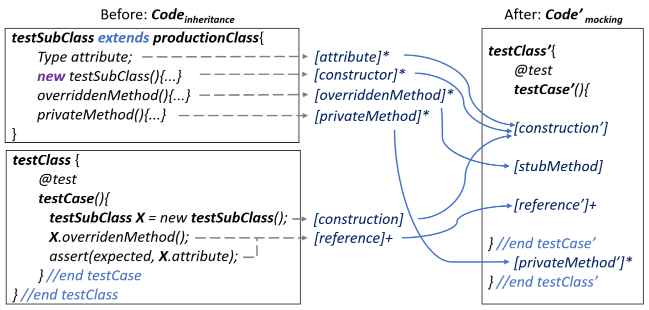
After the empirical study, the team summarized that 25% cases can be potentially refactored automatically by excluding complicated cases which require case-by-case solutions. Then, after our tool implementation (introduced in Section 5), we applied the tool on the entire empirical study dataset and confirmed that these 25% cases can indeed be refactored fully automatically.

3.2 Findings

Table 1 shows the data from the empirical investigation. The first column ("Proj.") lists the project names. The second column ("#SubCl.") indicates the total number of test subclasses in a project. The third

Proj.	#SubCl.	Succ.	Others		
			Infeasible	Not-Auto	Exec. Iss.
Dubbo	148	43 (29%)	79 (53%)	12 (8%)	14 (9%)
Druid	291	66 (23%)	173 (59%)	12 (4%)	40 (14%)
Accumulo	161	46 (29%)	85 (53%)	21 (13%)	9 (6%)
Cayenne	151	40 (26%)	99 (66%)	10 (7%)	2 (1%)
CloudStack	81	13 (16%)	58 (72%)	3 (4%)	7 (9%)
Sum	832	208 (25%)	494 (59%)	58 (7%)	72 (9%)

Table 1: Manual Refactoring Datasets



```
testSubClass extends productionClass{
    Type attribute;
    new testSubClass(){...}
    overriddenMethod(){...}
    privateMethod(){...}
}
```

```
testClass {
    @test
    testCase(){
      testSubClass X = new testSubClass();
      X.overridenMethod();
      assert(expected, X.attribute);
    } //end testCase
} //end testClass
```

```
testClass'{
    @test
    testCase'(){

      [construction']

      [stubMethod]

      [reference']+

    } //end testCase'
    [privateMethod']*
} //end testClass'
```

Figure 2: Illustration of Refactoring

column ("Succ.") shows the total number (percentage) of test subclasses that can be successfully replaced by using Mockito. That is, the refactoring of 208 (25%) cases from the five projects can be automated. The remaining cases are in three categories: ① The refactoring is not feasible, which is shown in column "Infeasible", with two general reasons. First, certain design features in the inheritance do not abide with the rationale of mocking. Second, there are detailed implementation and technical issues that prevent the refactoring using Mockito. ② Although manual refactoring is feasible, full automation of the refactoring is not possible (column "Not-Auto"). This is due to the complicated design of the inheritance that requires case-by-case understanding for refactoring. For example, some test subclass contains an inner class definition. Refactoring requires in-depth understanding of the inner class. ③ The refactoring is not successful due to issues with test execution (column "Exec. Iss."). In these cases, we either have issues executing related test cases; or the test behavior changes after refactoring for reasons that require case-by-case investigation. In summary, *a non-trivial portion (ranging from 16% to 29%) of test subclasses can be automatically refactored*. This motivates the design of our automated refactoring framework.

4 PROBLEM FORMALIZATION

Based on the 208 successful refactoring cases from the empirical study, we formalize the auto-refactoring problem. It is a conversion from the left side to the right side:

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

4.1 Before Refactoring: $code_{inheritance}$

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:

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

- $constructor$ 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 $testSubClass$.

The convention "[]*" (borrowed from regular expression) indicates there is zero or more of a design element. The upper-left code snippet illustrates an example $testSubClass$.

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: 1) *construction*, which invokes a $constructor$ of $testSubClass$ to create an instance; and 2) *reference*, which accesses the attributes or call the methods of the instance. The lower-left code snippet of Figure 2 illustrates a simple example.

4.2 After Refactoring: $code'_{mocking}$

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 1) $[construction']$ to create a mock object of the $productionClass$, which replaces the instance created by $[construction]$ in $testCase$; 2) $[stubMethod]$, which replaces the $[overridenMethod]$ in $testSubClass$; and 3) $[reference']$ to the mock object, which replaces the respective $[reference]$ to the $testSubClass$ instance in $testCase$. We will explain the formal refactoring procedure in Section 5.2.

5 REFACTORING FRAMEWORK

The auto-refactoring framework, implemented as an Eclipse-plugin, addresses the above formalization with two components: 1) identifying refactoring candidates and 2) refactoring each candidate.

ID	Filter Criterion (What condition to look for?)	Rationale (Why is it not feasible/automatable?)
F-1.1	A *testSubclass* ***inherits multiple*** *productionClasses*.	This indicates that the *testSubclass* mocks multiple *productionClasses*. Mocking multiple *productionClasses* is not recommended [20].
F-1.2	A *testSubclass* ***overrides*** **JDK APIs**— particularly *hashCode()* or *equals()*.	*"Only mock types you own"* [21]—avoid mocking JDK APIs. Plus, mocking the two APIs will break *Mockito* since it is built upon them [22].
F-1.3	A *testSubclass* defines a **new *public* method**; this method is ***not*** in the *productionClasses*.	The *testSubclass* no longer "mocks" the *productionClasses*, when it contains extra, new behaviors.
F-1.4	A *testSubclass* has ***self-reference*** (e.g. declaring itself as an attribute).	Self-reference implies self-mocking—a mock object cannot mock itself.
F-1.5	A *testSubclass* contains a method that returns a ***generic type***.	The generic type indicates uncertain mocking behavior; a mock object should have certain behavior.
F-2.1	A *testSubclass* is ***not instantiated*** through its constructor.	The *testSubclass* is not used anywhere (no need to mock) or is instantiated through dynamic binding (thus cannot use Mockito).
F-2.2	A *testSubclass* contains special ***code annotations***	The annotations are mostly project specific or from a special library [23]. Mockito does not support them.
F-2.3	A *testSubclass* has external access to a ***protected attribute/method*** in the *productionClass*.	*Mockito* does not support the access to protected elements. This requires a more powerful framework, such as PowerMock [9]
F-3.1	A *testSubclass* instance is ***passed as a parameter*** across multiple test cases/methods in a *testClass*.	It requires manual effort to create a good test design, where a mock object and the related variables pass along test cases and methods.
F-3.2	A *testCase* creates and uses ***a collection*** (e.g. Set or Map) of *testSubclass* instances.	It requires manual effort to create a good test design with a collection of mock objects and the variables that associate with each mock object.
F-3.3	A *testSubclass* contains an ***inner class*** definition.	It requires manual understanding to properly refactor the inner class.

Table 2: Refactoring Candidate Identification-Filters

5.1 Refactoring Candidate Identification

Based on the empirical study, we construct a taxonomy of 11 exclusion criteria to automatically exclude test subclasses that do not match the problem formalization—i.e. those that are not feasible to refactor or the refactoring cannot be automated. In Table 2, we list the detailed filtering conditions (column 2), as well as the rationale of each filtering condition (column 3). For the sake of clarify, we organize the 11 filtering conditions in three general layers: 1) Layer1—*F-1.1* to *F-1.5*—which excludes cases that are not suitable for mocking; 2) Layer2—*F-2.1* to *F-2.3*—which excludes cases that cannot be refactored due to detailed implementation limitations with Mockito; and 3) Layer3—*F-3.1* to *F-3.3*—which excludes cases with complicated design, such that they are not appropriate to refactor automatically.

5.2 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. The overall rationale is to eliminate the usage of *testSubClass*, and replace it with a mock object, which preserves all the attributes and behaviors of the *testSubClass*. Towards this, 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 that replaces the *testSubClass*.

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

5.2.1 *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 sub-steps 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 [1].

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: 1) it is a *boolean* or *int*; 2) it is only read/written in a certain methods of the mock object; and 3) 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.

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

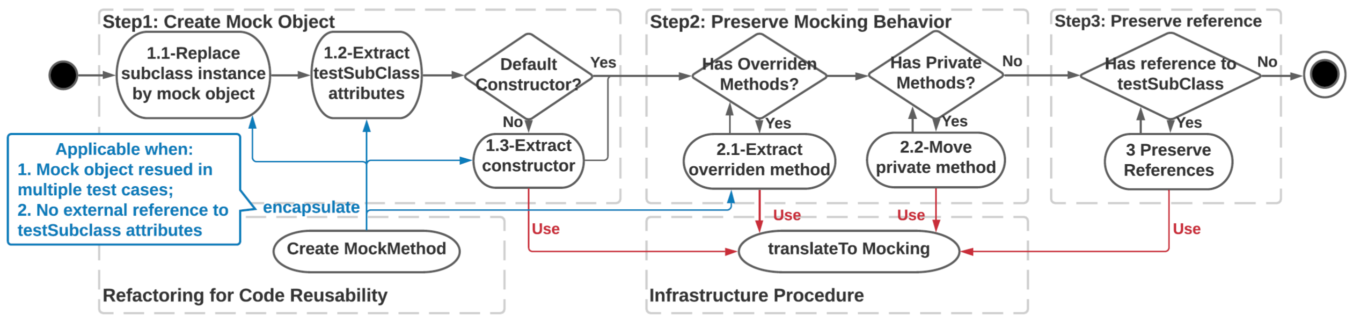
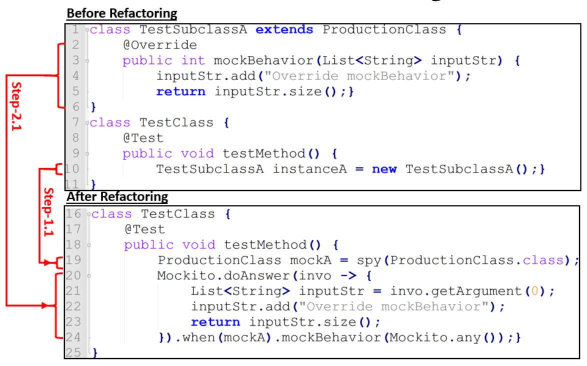

Step1: Create Mock Object
1.1-Replace subclass instance by mock object
1.2-Extract testSubClass attributes
Default Constructor?
Yes
No
1.3-Extract constructor
Step2: Preserve Mocking Behavior
Has Overriden Methods?
Yes
2.1-Extract overriden method
Has Private Methods?
No
Yes
2.2-Move private method
Step3: Preserve reference
Has reference to testSubClass
No
Yes
3 Preserve References
Applicable when:
1. Mock object resued in multiple test cases;
2. No external reference to testSubclass attributes
encapsulate
Create MockMethod
Refactoring for Code Reusability
Use
Use
Use
Use
translateTo Mocking
Infrastructure Procedure

Figure 3: Automated Refactoring Procedure



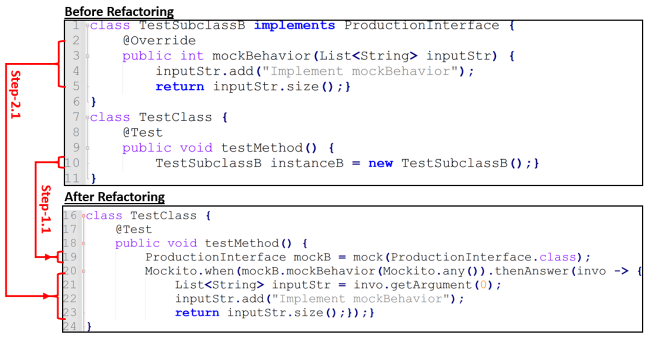
Before Refactoring

```
1  class TestSubclassA extends ProductionClass {
2      @Override
3      public int mockBehavior(List<String> inputStr) {
4          inputStr.add("Override mockBehavior");
5          return inputStr.size();}
6  }
7  class TestClass {
8      @Test
9      public void testMethod() {
10         TestSubclassA instanceA = new TestSubclassA();}
11 }
```

After Refactoring

```
16 class TestClass {
17     @Test
18     public void testMethod() {
19         ProductionClass mockA = spy(ProductionClass.class);
20         Mockito.doAnswer(invo -> {
21             List<String> inputStr = invo.getArgument(0);
22             inputStr.add("Override mockBehavior");
23             return inputStr.size();
24         }).when(mockA).mockBehavior(Mockito.any());}
25 }
```

(a) "Spy" and "doAnswer"



Before Refactoring

```
1  class TestSubclassB implements ProductionInterface {
2      @Override
3      public int mockBehavior(List<String> inputStr) {
4          inputStr.add("Implement mockBehavior");
5          return inputStr.size();}
6  }
7  class TestClass {
8      @Test
9      public void testMethod() {
10         TestSubclassB instanceB = new TestSubclassB();}
11 }
```

After Refactoring

```
16 class TestClass {
17     @Test
18     public void testMethod() {
19         ProductionInterface mockB = mock(ProductionInterface.class);
20         Mockito.when(mockB.mockBehavior(Mockito.any()).thenAnswer(invo -> {
21             List<String> inputStr = invo.getArgument(0);
22             inputStr.add("Implement mockBehavior");
23             return inputStr.size();});}
24 }
```

(b) "Mock" and "thenAnswer"

Figure 4: Mock Object Creation and Stub Method

The other type of attributes 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 5.2.4.

5.2.2 *Step2-Preserve Mocking Behavior:*

This preserves the mocking behaviors by treating the *overriddenMethods* and *privateMethods* 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* adds **additional** actions to the stubbed method [24]. It ensures *type safe* thus should be preferred whenever possible. While, *doAnswer* entirely replaces the original method behavior [25], 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. In addition, there are specific method stubbing syntax for different kinds of behaviors, summarized here [1]. For example, *doReturn* and *thenReturn* are for stubbing methods that just return certain objects.

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 reference to the *testSubClass* attributes/methods, we also need to use *translateToMocking* procedure to convert the syntax before moving.

Step-2.2: Move each private method from *testSubClass* to *testClass'*. These method movements cannot be directly copy-and-paste 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*.

5.2.3 *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 executed 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 $testCase'$.

5.2.4 *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, referrence statements—as input, and convert them follow the syntax after refactoring.

For each statement, stm, in the input code body, this procedure makes the following conversions: ① If stm has reference to an attribute in *testSubclass*, 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. 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 $testCase'$—or the attribute in the $testClasss'$—depending on where the attribute is extracted in step 1.2. ② If stm contains reference to a $privateMethod$ in the $testSubClass$, we replace the reference to be the $privateMethod'$ in $testClass'$. ③ 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 $testCase'$.

5.2.5 *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.

5.3 Implementation

We implemented the auto-refactoring framework as an Eclipse-plugin [1]. It works in two steps: 1) Identifying eligible 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, and click the *"Detect Mock Refactor Candidates"*. The identification relies on the *AST Parser* of *Eclipse JDT* [26] to filter out cases that defined in Section 5.1 based on the detailed code syntax. 2) Selecting cases and performing refactoring. The tool will notify users the list of identified refactoring candidates (i.e. sub-classes). The user needs to select a candidate to proceed with the refactoring, by clicking on *"Start Refactor"*. The implementation of refactoring relies on the *ASTRewrite* mechanism of the *Eclipse JDT* [26].

6 EVALUATION

6.1 Evaluation Dataset

We select four new projects, with a total of 610 test subclasses. They are: JackRabbit—an open source content repository for the Java platform [27], Log4J2—a Java-based logging utility [28], Qpid-Proton-J—a high-performance, lightweight messaging library [29], Apache Commons—which focuses on all aspects of reusable Java Components, with 40 subprojects, including Commons-Collections, Commons-Lang, Commons-Logging, etc. 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 3.

6.2 Research Questions and Rationale

RQ1 to RQ4 are answered by quantitative evaluation; RQ5 is answered by qualitative evaluation.

- *RQ1: How generally applicable is the refactoring framework?* We report the number and percentage of test subclasses that are filtered out in the identification, and that are successfully refactored.
- *RQ2: Do the test behaviors remain consistent before and after the refactoring with injected mutations?* We inject mutations[30] to the production code that may impact the refactored test cases. We then investigate the execution status of each mutant—survived or killed—before and after the refactoring. If a survived/killed mutation before refactoring still survives/is killed after the refactoring, it suggests that the test case behaviors remain consistent.
- *RQ3: How does the refactoring affect code complexity?* We report the code complexity measured by the LOC, #methods, and #fields, as well the amount of dependencies from test-to-production code and among the test code.
- *RQ4: What is the performance of our refactoring framework?* We report the execution times in the candidate identification and the refactoring implementation, respectively.
- *RQ5: How is the quality of the auto-refactoring solution in real-developers' opinion? And how does it compare to the manual refactoring solution implemented by real-developers?* We conduct a user study involving full-time developers to both manually implement refactoring and review the refactoring solutions generated by our framework. The goal is to understand the value, benefits, and quality of our refactoring solution, especially when compared to manual solution created by experienced developers.

6.3 Quantitative Evaluation Results (RQ1-RQ4)

RQ1: In Table 3, in all four projects (row "Total"), there are totally 610 test sub-classes (column "#SubCl."). Among these, 217 (column "F-1"), 50 (column "F-2"), and 86 (column "F-3") cases are filtered out by the three filtering layers (Section 5.1). Therefore, there are 257 (42%) (column "#Candidates") identified as feasible refactoring candidates. Furthermore, the refactoring of 27 (4%) test sub-classes leads to compile errors (column "Comp.") due to syntax issues that were not captured in the dataset of the empirical study. In addition, 16 (3%) test sub-classes, after the refactoring, lead to test behavior discrepancies from before the refactoring (column "Discre."). Discrepancies happen in special cases. For example, some test cases use the metadata of the test subclass at run-time. The test cases fail after the refactoring. Totally, 214 cases are successfully refactored.

Proj.	#SubCl.	Identification				Refactoring		
		F-1	F-2	F-3	#Candidates	Comp.	Discre.	Succ.
JackRabbit	71	15	4	9	43 (60%)	3 (4%)	0 (0%)	40 (56%)
Log4J2	100	31	19	17	33 (33%)	5 (5%)	3 (3%)	25 (25%)
Qpid-Proton-J	34	13	0	12	9 (26%)	0 (0%)	0 (0%)	9 (26%)
Commons	405	158	27	48	172 (42%)	19 (5%)	13 (3%)	140 (35%)
Total	610	217	50	86	257 (42%)	27 (4%)	16 (3%)	214 (35%)

Table 3: Applicable Auto-Refactoring in Testing Datasets

Summary: Our approach successfully refactors 214 cases—indicating a 83% (214/257) successful rate over the feasible cases and 35% (214/610) successful rate over all 610 test sub-classes. This suggests good applicability of our approach.

RQ2: We use PIT [31], a state-of-the-art mutation testing system, to generate mutators. Table 4 summarizes all the generated default mutators [32] and their percentage in our dataset.

Mutator	Description	%
Negate Conditions	mutate all conditions by its logical negation	33%
Void Method Calls	remove method calls to void methods	26%
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%
Primitive Returns	replace int, short, long, char, float and double return values with 0	4%
Conditionals Boundary	replaces the relational operators with their boundary counterpart	4%
Increments	replaces increments with decrements and vice versa	1%

Table 4: Applicable Default Mutators by Pitest

The execution of the mutations are reported in Table 5. Column 2 shows the number of refactored test cases executing the mutants, ranging from 126 (Qpid) to 2510 (Commons). Column 3 shows the number of successfully mutated production classes by PIT, accounting for 35% to 56% 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 4) cannot be applied. Column 4 shows that a total of 51811 mutants—from 1156 (Qpid) to 35902 (Commons) in each project—are generated.

Proj.	#T.Cs	#P.(%M)	#Mut.	Covered		Killed		Survived	
				Before	After	Before	After	Before	After
JackRabbit	431	107 (47%)	9730	3803 (39%)	✓	2981	+1, -2	822	+2, -1
Log4J2	424	152 (35%)	5023	1548 (31%)	✓	992	✓	556	✓
Qpid-Proton-J	126	43(51%)	1156	332 (29%)	✓	298	✓	34	✓
Commons	2510	700(56%)	35902	17120 (48%)	+3	12709	+121, -120	4411	+123, -121
Total	3491	1002 (50%)	51811	22803 (44%)	+3	16980	+122, -122	5823	+125, -122

Table 5: Mutation Status Change

Column 5 and 6 report the coverage statistics of executing the mutants before and after the refactoring. Among all mutants, 332 (29%) to 17120 (48%) are covered before applying our refactoring approach. The coverage on the mutants remains highly consistent after the refactoring, except for 3 (out of totally 51811) mutants. The discrepancy on these 3 mutants needs further investigation. Note that not all mutants are covered since we use the test cases supplied by the projects, which do not achieve 100% coverage. In addition, we only execute test cases affected by the refactoring.

Among the covered mutants, we observe (Column 7 — column 10) that, except Commons and JackRabbit, the mutation status (Killed or Survived) remains consistent before and after refactoring. There is a total of 247 mutants changed status in Commons — 122 changed from survived to killed and 125 changed from uncovered (3 mutant) or killed (122 mutants) to survived. Similarly, in JackRabbit, 3 mutants change the status after the refactoring. 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 coverage of the mutants before and after applying our approach is highly consistent ($< 0.001\%$ of the generated mutants changed coverage). The test cases, after refactoring using our approach, consistently cover, kill, or survive with 99% of the 51,811 mutants injected into the production code. This indicates that our approach generally preserves test behaviors in term of detecting defects (in the form of mutants).

RQ3: We investigate the complexity in two main aspects: 1) the basic complexity metrics, including the LOC, #Methods, and #Fields; and 2) the coupling from test to production code, and that among the test code. Table 6 shows the study results.

Proj.	Avg. (%Increase)			# (%In.) of T-P Dps.		# (%In.) of T-T. Dps.	
	LOC	#Method	#Field	Inherit.	Regular	Inherit.	Regular
JackRabbit	353 (4%)	64 (-5%)	6 (-9%)	37 (-38%)	225 (-12%)	329 (-8%)	956 (-0%)
Log4J2	108 (8%)	17 (-11%)	4 (-20%)	100 (-24%)	445 (-10%)	16 (0%)	75 (-7%)
Qpid-Proton-J	178 (6%)	28 (-14%)	6 (-22%)	34 (-26%)	264 (-9%)	22 (0%)	75 (-7%)
Commons	261 (4%)	29 (-5%)	7 (-11%)	399 (-35%)	4010 (-5%)	168 (-1%)	737 (-9%)

Table 6: Code Complexity Change

Columns 2 – 4 report the average number of lines of code, methods, and Fields before the refactoring and their increments after the refactoring over all refactoring cases. A negative percentage indicates the measure decreases. We observe that after the refactoring, 1) the LOC averagely increases 4% to 8% for each refactoring case; and 2) both the number of methods and the number of fields decreases, by 5% to 14% and 9% to 22% respectively. The LOC increases due to cases where the reusable *mock method* (Section 5.2.5) is not applicable. The number of methods and fields decreases due to Step 1.2 and Step 2.1 (Section 5.2).

Columns 5–6 report the test-to-production coupling in terms of the number of dependencies before the refactoring and its increments after the refactoring (column "#(%In.) of T-P Dps."). Here, we separate the inheritance (column "Inherit.") and other general dependencies (column "Regular"). We observe that: 1) the number of test-to-production inheritance non-trivially decreased by 24% (Log4J2) to 38% (JackRabbit); and 2) the number of test-to-production regular dependencies decreased by up to 12% (JackRabbit).

In columns 7—8, we examine the number of dependencies among the test classes and their increments after the refactoring (column "#(%In.) of T-T Dps."). Similarly, we separate the inheritance relationship and any other dependencies. We observe that: 1) the number of inheritance relationship among test classes decreased in JackRabbit and Commons, by 8% and 1%, respectively, and remained stable in the other two projects; and 2) the regular dependencies among test classes decreased by 7% or 9% in projects except JackRabbit.

Summary: The refactoring overall decreases code complexity. In particular, it non-trivially decouples the test from the production code, by removing 24% to 38% of the inheritance and 5% to 12% 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 9%. Meanwhile, it slightly increases the LOC (4% to 8%) of the refactored test classes, but more obviously decreases the number of methods (up to 14%) and fields (up to 22%).

RQ4: Table 7 shows the running time (in seconds) of the framework. Column "#TestCl." shows the total number of test classes in each project—131 (Qpid) to 3599 (Commons). This is the initial input size to the candidate identification. Among these, we identify 34 (Qpid) to 405 (Commons) test subclasses, which are processed by the three filtering layers (Section 5.1). The entire detection time ranges from 30 to 250 seconds. The average execution time of refactoring each case ranges from 0.7 (JackRabbit) to 1.5 (Log4J2) seconds, with the standard deviation of 0.4 (Qpid) to 2.2 (Log4J2).

Proj.	Detection Time (s)			Refactoring Time (s)				
	#TestCl.	#SubCl.	Total-T	#Case	Avg-T	Max-T	Min-T	Std
JackRabbit	1060	71	42	26	0.7	2.2	0.3	0.6
Log4J2	1069	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	3599	405	250	139	1.0	11.4	0.2	1.3
Total	5859	610	535	200	1.0	11.4	0.2	1.3

Table 7: Auto-refactoring Performance

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.

6.4 Qualitative Evaluation (RQ5)

6.4.1 ***Study Design:*** **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 4 and all possible refactoring steps introduced in Section 5.2. 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. After refactoring/reviewing a case, the participant takes a survey to evaluate the value, the quality (only for review case), and the benefits of refactoring. As such, each case is manually refactored by three participants, and the respective auto-refactoring solution is reviewed by another three participants. Thus, each case receives six survey results—three for the manual refactoring and three for the provided auto-refactoring solution—from all six participants. This minimizes individual biases of participants.

The study is held remotely on the AWS servers [33]. 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 8, 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.**

	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

Table 8: Successful Manual Refactoring

Survey Questions: 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 focuses on assessing the quality of the auto-refactoring solutions. SQ2 assesses the general benefits of refactoring. SQ3 and SQ4 evaluate the detailed benefits (or drawbacks) of the refactoring.

For SQ2 to SQ4, we also investigate the discrepancies between implement and review cases to show how the auto-refactoring solution is different from the manual refactoring.

*6.4.2 **Study Results:*** **SQ1 (R):** Figure 5 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.

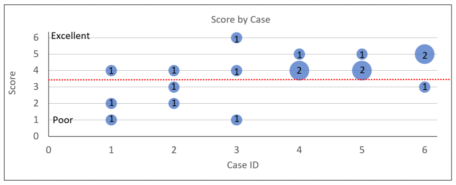


Figure 5: SQ1: Quality of Auto-Refactoring Solution

We investigate the comments from the participants. We find that the the negative scores fall into three types: 1) The criticism is about the original test design, which is irrelevant to the Mockito-based refactoring (Case 1, case 3, and case 6). 2) 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 5.1). However, the participant thinks that we should also refactor it. The other misunderstanding is that the participant suggests using *mock method* (Section 5.2.5) when it is not appropriate. 3) 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 6, 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 1) participants expect to see improvements on the test design/logic itself; and 2) participants preference to separate the mock behaviors in a subclass. **Overall, participants agree that using mocks to replace inheritance improves the code quality.**

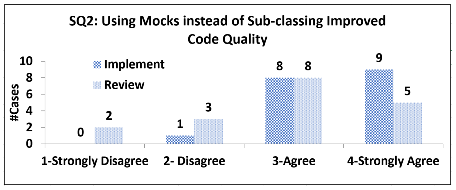


Figure 6: SQ2: Improved Code Quality (Impl. vs. Review)

SQ3 (I&R): Table 9 summarizes ratings on the refactoring benefits in cohesion/concise, 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.

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%

Table 9: Refactoring Benefits (Implementation vs. Review)

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, compared to the implementation cases. This indicates that the manual refactoring implementation by developers boosts these benefits on more cases and to a higher degree.

To further investigate the causes of the discrepancies, we reviewed the code difference between the manual refactoring and the auto-refactoring solutions. We find that the discrepancies are largely due to the improvements participants made to the test logic and design in the implementation set, such as simplifying the test logic and removing redundancy. **In conclusion, our auto-refactoring framework alone helps developers reap the three benefits to some extent. To achieve further benefits, developers need to improve the test logic 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.**

Potential Improvements: Finally, we compare the auto and manual refactoring solutions. This 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) A obvious draw-back of our framework is that the original code comments cannot be retained after refactoring.

7 LIMITATIONS AND THREATS TO VALIDITY

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 [34]. In this work, we investigate this problem in a qualitative study involving real developers (RQ5) 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

Our framework has five 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 51,811 mutants to evaluate the test behaviour preservation, 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.

An 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 6.4.

8 RELATED WORK

Software Refactoring: Significant software development cost is devoted to software maintenance [35–38], as software becomes more complex and drifts away from its original design [39–42]. Refactoring is an important maintenance activity that restructures a system and improves code quality [41, 43, 43–49]. Kim showed that refactoring is challenging and there generally lacks tool support [50]. In past years, researchers proposed methods and tools to automate the refactoring process [51–62]. Most refactoring tools focus on detecting and refactoring God Classes [46, 56, 63–65], and eliminating Code Clone [52–54, 66–68]. Tsantalis et. al proposed a refactoring approach to replace state checking (i.e. *if/else*) with polymorphism to reduce code complexity [57]. **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 [46]. Test smells are the sub-optimal design choices in test code [69, 70]. They can make test cases less effective and more difficult to understand [69, 71–76]. There are various techniques and tools to support automated test smell analysis [69, 77–83]. Van Deursen et. al defined a catalog of 11 test smells [69]. Based on this catalog, Van Rompaey et. al introduced a metric-based technique to identify two smells, *General Fixture* and *Eager Test* [79]. Greiler et. al developed a Maven plugin to detect test fixture related smells and provide guidance for refactoring them [80]. Santana et. al implemented an Eclipse plugin to refactor *Assertion Roulette* and *Duplicate Assert* [82]. Other works foucs on analyzing the impact of test smells [71, 84–86]. **To our best knowledge, no prior work investigated sub-optimal practice in unit test mocking.**

9 CONCLUSION

We proposed a refactoring framework and implemented it as an Eclipse-Plugin 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 *four* 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. 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.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[2] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.
[3] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.
[4] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.
[5] 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.
[6] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. Mock objects for testing java systems. *Empirical Software Engineering*, 24(3):1461–1498, 2019.
[7] https://easymock.org/.
[8] https://site.mockito.org/.
[9] https://powermock.github.io/.
[10] https://junit.org/junit5/.
[11] https://wiki.python.org/moin/PyUnit.
[12] 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.
[13] 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.
[14] Jeff Grigg. http://wiki.c2.com/?ArrangeActAssert/, 2012.
[15] https://dubbo.apache.org/.
[16] https://druid.apache.org/.
[17] https://accumulo.apache.org/.
[18] https://cayenne.apache.org/.
[19] https://cloudstack.apache.org/.
[20] https://javadoc.io/static/org.mockito/mockito-core/3.1.0/org/mockito/MockSettings.html#extraInterfaces-java.lang.Class...-.
[21] 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.
[22] https://github.com/mockito/mockito/wiki/FAQ#what-are-the-limitations-of-mockito.
[23] https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations.
[24] https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/stubbing/OngoingStubbing.html#thenAnswer-org.mockito.stubbing.Answer-.
[25] Java Code Geeks. Mockito programming cookbook. https://www.javacodegeeks.com/wp-content/uploads/2016/09/Mockito-Programming-Cookbook.pdf.
[26] https://projects.eclipse.org/projects/eclipse.jdt.
[27] https://jackrabbit.apache.org/jcr/index.html.
[28] https://logging.apache.org/log4j/.
[29] https://qpid.apache.org/.
[30] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.
[31] https://pitest.org/.
[32] https://pitest.org/quickstart/mutators/.
[33] https://aws.amazon.com/lightsail/.
[34] 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.
[35] 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.
[36] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE transactions on software engineering*, 25(4):493–509, 1999.
[37] 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.
[38] 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.
[39] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.
[40] Tor Guimaraes. Managing application program maintenance expenditures. *Communications of the ACM*, 26(10):739–746, 1983.
[41] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.
[42] 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.
[43] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, 2020.
[44] William F Opdyke. Refactoring object-oriented frameworks. 1992.
[45] William C Wake. *Refactoring workbook*. Addison-Wesley Professional, 2004.
[46] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
[47] 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.
[48] 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.
[49] 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.
[50] 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.
[51] 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.
[52] 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.
[53] 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.
[54] 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.
[55] George Ganea, Ioana Verebi, and Radu Marinescu. Continuous quality assessment with incode. *Science of Computer Programming*, 134:19–36, 2017.
[56] 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.
[57] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of refactoring opportunities introducing polymorphism. *Journal of Systems and Software*, 83(3):391–404, 2010.
[58] 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.
[59] 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.
[60] Liming Zhao and Jane Huffman Hayes. Rank-based refactoring decision support: two studies. *Innovations in Systems and Software Engineering*, 7(3):171–189, 2011.
[61] 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.
[62] 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.
[63] 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.
[64] 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.
[65] 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.
[66] 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.
[67] 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.

[68] 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.

[69] 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.

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

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

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

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

[79] 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.

[80] 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.

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

[82] 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.

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

[84] 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.

[85] 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.

[86] 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.