# JMocker: Refactoring Test-Production Inheritance by Mockito

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

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

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

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

Sunny Wong
Sunny@computer.org
Analytical Graphics, Inc.
USA

## ABSTRACT

Mocking frameworks are dedicated to creating, manipulating, and verifying the execution of "faked" objects in unit testing. This helps developers to overcome the challenge of high inter-dependencies among software units. Despite the various benefits offered by existing mocking frameworks, developers often 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. We contribute a refactoring tool as an Eclipse Plugin, named *JMocker*, to automatically identify and replace the usage of inheritance by using Mockito—a well received mocking framework for Java projects. We evaluate *JMocker* on *four* open source projects and successfully refactored 214 cases in total. The evaluation results show that our framework is efficient, applicable to different projects, and preserves test behaviors. According to the feedback of six real-life developers, *JMocker* improves the design quality of test cases. *JMocker* is available at https://github.com/wx930910/JMocker. The tool demo can be found at https://youtu.be/HFoA2ZKCoxM.

## 1 INTRODUCTION

A key challenge to unit testing is that software elements are inter-dependent on each other—as such the testing of software units depends on each other [4, 11]. This hinders the developers to efficiently test the system as true "units". To overcome this challenge, practitioners proposed the concept of mocking to achieve test dependency isolating. That is, they isolate the core function under test (*FUT*) from its dependencies by replacing the dependencies as "faked" objects [12, 13]. For instance, the *FUT* may depend on an external server that is not deployed. Instead of waiting for the deployment of the server, developers create a "faked" server that provides the dummy functions as intended for the testing purposes.

Mocking frameworks, such as *easyMock* and *Mockito* provide powerful functions for easily creating mock objects, controlling their behavior, and verifying the execution/status of the mock objects. However, in practice, developers often turn to a "hand-rolled" approach—inheritance—for mocking [14]. That is, to create a "fake" object, developers create a subclass of the dependent production class, and control the subclass's behavior through method overriding. The problem with sub-classing for mocking is that it is not intended for mocking. Doing so will compromise the design quality of unit tests. For instance, using inheritance for mocking may lead to the following drawbacks, compared to using a mocking framework: 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. Consequently, using inheritance for mocking may compromises the understandability and maintainability of test cases in the long run [10, 14].

Prior work developed a variety of techniques that enable unit test refactoring to improve code readability [5] and code quality [6, 9]. However, no existing work has focused on improving unit test design by refactoring the usage of inheritance into mocking frameworks. In this paper, we present an automated refactoring tool as an Eclipse plugin, named *JMocker*. It first searches the code base of a project, and identifies feasible refactoring candidates by filtering out infeasible sub-classing cases using 11 criteria that we summarize from empirical experience. Next, it automatically performs the refactoring on certain refactoring candidate(s) or batch process all refactoring candidates, based on the user's selection. Finally, the user may choose to view the generated refactoring solution in a diff-view, showing the side-by-side comparison of the before and after refactoring, similar to the diff view provided by Git. This allows the users to review the refactoring solution and make changes to it, if necessary. As evaluation, we successfully apply *JMocker* on four real-life projects, running efficiently. The refactoring solutions preserve the test behavior, decouple test code from the production code, and 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.

## 2 A MOTIVATING EXAMPLE

We use an example to illustrate and compare the difference between mocking through inheritance and through Mockito:

In a course management system, *CourseRegistrationService* defines a service, *registerCourse*, to register courses for students. This service depends on another class, *DatabaseService*, which is


*ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA*

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516836

responsible for managing the course database and sending out registration confirmations. The method, *register*, in *DatabaseService* first registers a course for the student; once registered, it sends out the registration confirmation to the student. Another method, *sendConfirmation*, sends confirmation through an external server. The *FUT* is the *registerCourse* in *CourseRegistrationService*. The problem is that the dependency of *FUT*, *DatabaseService*, is not fully implemented yet—neither the database nor the external service is available. Thus, we isolate the *FUT*, *registerCourse*, from its dependency, *CourseRegistrationService*, by mocking the latter.

*Mocking by Inheritance.* In Figure 1a, *MockDatabaseService extends* the *DatabaseService* (line 1). The former mocks the behaviors–*register* and *sendConfirmation*—of the latter through method overriding. Two new private attributes, *registered* (line 2) and *num* (line 3), are defined for tracking the execution of the two overridden methods. That is, *registered* is set to be *true* (line 6) when *register* executes; while *num* increments (line 10) each time *sendConfirmation* executes.

The test case, *testRegisterCourse*, follows the "AAA" (Arrange, Act, Assert) pattern [7]. First, it arranges the environment for testing. This includes creating an instance of *MockDatabaseService*—*databaseService*—(line 15), creating an instance of *CourseRegistrationService*, *courseRegistrationService*, which is the *FUT* and set *databaseService* for *courseRegistrationService*. Of particular note, since the logic defined in this subclass prepares mocking behaviors for the unit test case, it is also part of the "Arrange" in the "AAA" pattern. Next, it acts the *FUT* (line 18). Lastly, the test case asserts the value of *registered* and *num* with *MockDatabaseService* (line 19 and 20). They confirm that *registered* is *true*, indicating method *register* is executed; and that *num* equals 2, indicating that one confirmation are sent.

```
1  class MockDatabaseService extends DatabaseService {
2      private boolean registered = false;
3      private int num = 0;
4      @Override
5      public boolean register(String courseID, String studentID) {
6          registered = true;
7          sendConfirmation(courseID, studentID);
8          return true;}
9      @Override
10     public void sendConfirmation(String courseID, String studentID) {num++;}
11 }
12 class TestCourseRegistrationService {
13     @Test
14     public void testRegisterCourse() {
15         MockDatabaseService databaseService = new MockDatabaseService();
16         CourseRegistrationService courseRegistrationService = new CourseRegistrationService();
17         courseRegistrationService.setDatabaseService(databaseService);
18         courseRegistrationService.registerCourse("courseID", "studentID");
19         assertTrue(databaseService.registered);
20         assertEquals(1, databaseService.num);
21     }
22 }
```

Arrange / Subclass — Arrange — Act — Assert

**(a) Mocking by Inheritance**

```
24 class TestCourseRegistrationService {
25     @Test
26     public void testRegisterCourse() {
27         DatabaseService databaseService = mock(DatabaseService.class);
28         Mockito.when(CourseDatabaseService.register("courseID", "studentID"))
29         .thenAnswer(invo -> {
30             databaseService.sendConfirmation("courseID", "studentID");
31             return true;
32         });
33         CourseRegistrationService courseRegistrationService = new CourseRegistrationService();
34         courseRegistrationService.setDatabaseService(databaseService);
35         courseRegistrationService.registerCourse("courseID", "studentID");
36         Mockito.verify(databaseService, Mockito.atLeastOnce()).register("courseID", "studentID");
37         Mockito.verify(databaseService, Mockito.times(1)).sendConfirmation("courseID", "studentID");
38     }
39 }
```

Mock sendEmail — Arrange — Act — Assert

**(b) Mocking by Mockito**

**Figure 1: A Motivating Example**

*Mocking by Mockito.* In Figure 1b, Mockito directly creates a "mock" of the *DatabaseService* (line 27). Mockito offers a light-weighted method stubbing for controlling the behaviors of the mock object for testing purposes. The goal is to avoid the real execution of *DatabaseService* (dependency) and focus on its interactions with *registerCourse* (the *FUT*). Thus, in line 28-32, we stub the mocking behavior of *DatabaseService* when *register* is invoked. The *sendConfirmation* should *do nothing*, since we want to avoid sending real confirmations. Thus, there is no need to stub it. Acting the *FUT* (line 35) remains the same as using inheritance. Mockito also provides an explicit mechanism for verifying the behaviors/status of the mock objects. In line 33 and 34, we directly *verify* the execution of *register* and *sendConfirmation*.

*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. 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. 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. 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 cases. 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.

## 3 JMOCKER APPROACH AND IMPLEMENTATION

As shown in Figure 2, *JMocker* works in three steps: 1) identifying the refactoring candidates, 2) refactoring each candidate, and 3) displaying the diff view of the before and after refactoring.

### 3.1 Identifying the Refactoring Candidates

*JMocker* first identifies cases of test-production inheritance for mocking, in particular those that are feasible for refactoring by using Mockito. This step excludes test subclasses that are not feasible to be replaced by using Mockito. Based on a large-scale empirical study of 832 test-production inheritance cases from 5 real-life projects [14], we establish a total of 11 excluding criteria. These excluding criteria work in three layers: 1) excluding cases that are not suitable for mocking, i.e. a test subclass inherits multiple production classes; 2) excluding cases that cannot be refactored due to limitations of Mockito, i.e. a test subclass contains special code annotations; and 3) excluding cases with complicated design, such that they are not appropriate to refactor automatically, i.e. a test class contains an inner class definition. Due to space limit, the details of the 11 criteria can be found here [14].

As shown in Figure 3a, after loading a project in Eclipse, a user first selects the scope, e.g. the entire project, a package, or a group

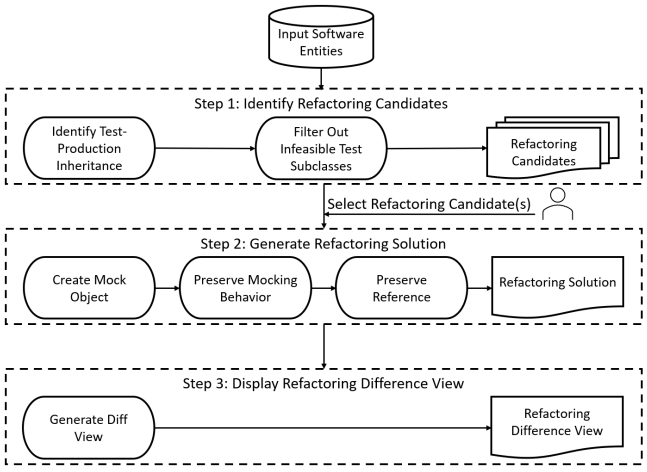
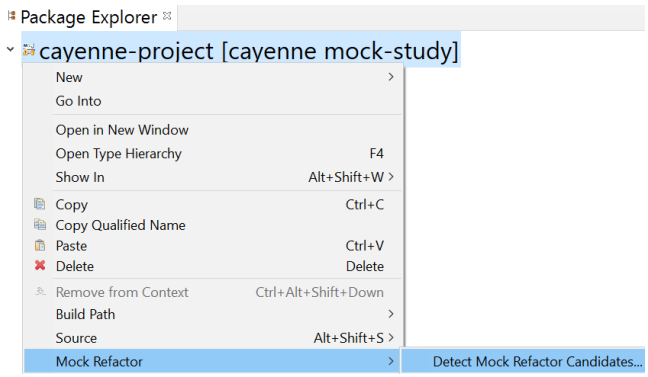
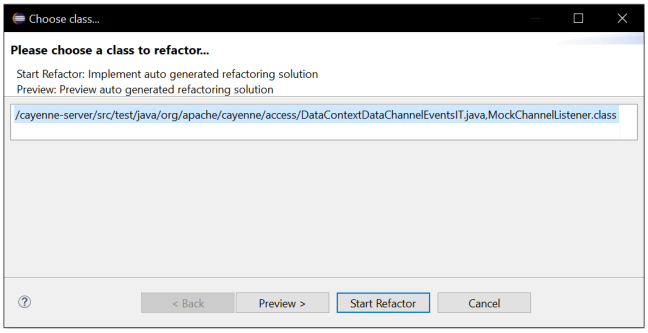



**Figure 2: Approach Overview**

of files, from which refactoring candidates should be identified. Here, we select project *Cayenne-project* and click the *"Detect Mock Refactor Candidates"*. *JMocker* will notify users with a list of identified refactoring candidates (i.e. classes), as shown in Figure 3b. The user needs to select a candidate to proceed with the refactoring, by clicking on *"Start Refactor"*.




**(a) Detect Refactoring Candidate**




**(b) Select Refactoring Candidate**

**Figure 3: Eclipse-Plugin**

## 3.2 Performing the Refactoring

The second step performs the refactoring to replace the usage of inheritance by using Mockito. This is a conversion from $code_{inheritance}$ before the refactoring to $code'_{mocking}$ after the refactoring.

*Before Refactoring.* A refactoring candidate $code_{inheritance}$ can be denoted as a triad: $code_{inheritance} =< testSubClass, productionClass, testClass >$.

Here, *testSubClass* extends the *productionClass*. It further consists of four key elements:

- *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 *testClass* leverages *testSubClass* to assist testing. It 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.

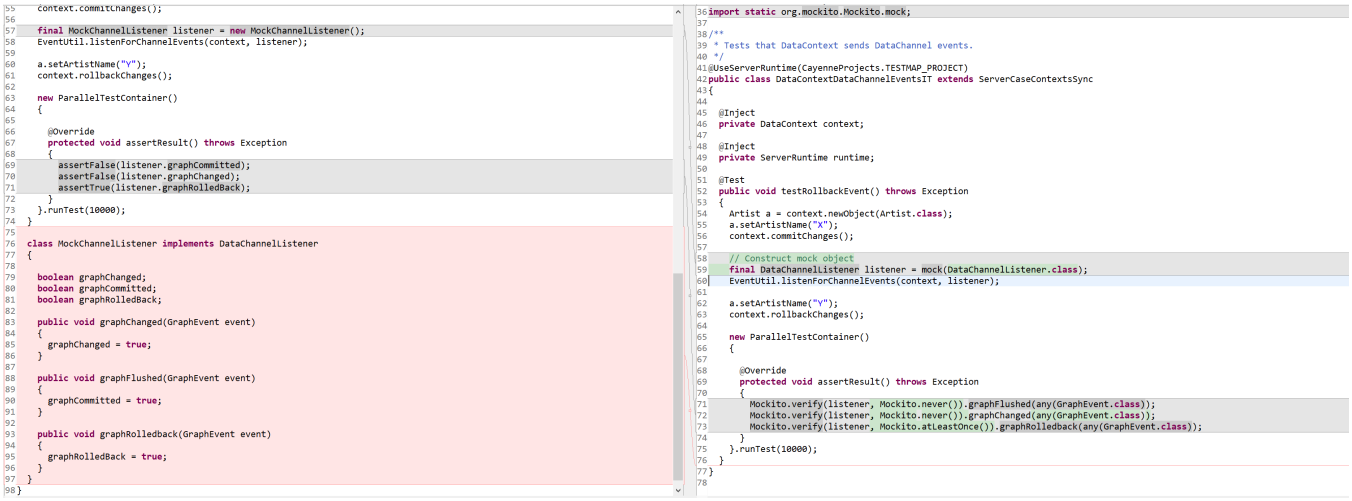
*After Refactoring.* The $code'_{mocking}$ eliminates *testSubClass* and replaces it by a mock object. As such, $code'_{mocking} =< production Class, testClass' >$. Thus *testClass* becomes *testClass'*, and each *testCase* in it becomes *testCase'*. Each refactored *testCase'* is consisted of 1) $construction'$ 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*.

*Auto-Refactoring Procedure.* The overall rationale of converting $code_{inheritance}$ to $code'_{mocking}$ is to eliminate the usage of *testSubClass*, and replace it with a mock object. The key is to preserve the behaviors of the *testSubClass*.

The refactoring process contains five logical parts:

**Part 1—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-parts are performed: 1) Replace *testSubClass* instance creation by mock object creation; 2) Extract the *attributes* of *testSubClass* to *testClass'*. This ensures that the status of the *testSubClass* instance is preserved for the mock object. And, 3) Extract the constructor logic from *testSubClass* to $construction'$. This ensures that the mock object has equivalent initial status as the *testSubClass* instance. 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.

**Part 2—Preserve Mocking Behavior:** This preserves the mocking behaviors by replacing the *overriddenMethods* in the *testSubClass*, by the *stubMethod* of Mockito which directly associates with the mock object created/used in $testCase'$. For example, in Figure 1a, *thenAnswer* stub (between line 27 to line 29) is used to replace *subscribe*() (between line 4 to line 8) in Figure 1b. Note that if the internal logic has reference to the *testSubClass* attributes/methods,



```
    context.commitChanges();

    final MockChannelListener listener = new MockChannelListener();
    EventUtil.listenForChannelEvents(context, listener);

    a.setArtistName("Y");
    context.rollbackChanges();

    new ParallelTestContainer()
    {

      @Override
      protected void assertResult() throws Exception
      {
        assertFalse(listener.graphCommitted);
        assertFalse(listener.graphChanged);
        assertTrue(listener.graphRolledBack);
      }
    }.runTest(10000);
  }

  class MockChannelListener implements DataChannelListener
  {

    boolean graphChanged;
    boolean graphCommitted;
    boolean graphRolledBack;

    public void graphChanged(GraphEvent event)
    {
      graphChanged = true;
    }

    public void graphFlushed(GraphEvent event)
    {
      graphCommitted = true;
    }

    public void graphRolledback(GraphEvent event)
    {
      graphRolledBack = true;
    }
  }
}
```

```
import static org.mockito.Mockito.mock;

/**
 * Tests that DataContext sends DataChannel events.
 */
@UseServerRuntime(CayenneProjects.TESTMAP_PROJECT)
public class DataContextDataChannelEventsIT extends ServerCaseContextsSync
{

  @Inject
  private DataContext context;

  @Inject
  private ServerRuntime runtime;

  @Test
  public void testRollbackEvent() throws Exception
  {
    Artist a = context.newObject(Artist.class);
    a.setArtistName("X");
    context.commitChanges();

    // Construct mock object
    final DataChannelListener listener = mock(DataChannelListener.class);
    EventUtil.listenForChannelEvents(context, listener);

    a.setArtistName("Y");
    context.rollbackChanges();

    new ParallelTestContainer()
    {

      @Override
      protected void assertResult() throws Exception
      {
        Mockito.verify(listener, Mockito.never()).graphFlushed(any(GraphEvent.class));
        Mockito.verify(listener, Mockito.never()).graphChanged(any(GraphEvent.class));
        Mockito.verify(listener, Mockito.atLeastOnce()).graphRolledback(any(GraphEvent.class));
      }
    }.runTest(10000);
  }
}
```

**Figure 4: Refactoring Diff View**

we need to use *translateToMocking* procedure to convert the syntax before moving.

**Part 3—Preserve Reference:** 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′*.

**Part 4—Create *MockMethod* (for Code Reusability):** A *testSubClass* could be created and used in multiple *testCases*. For each constructor in *testSubClass*, the refactored code block from Part-1 for mock object creation can be reused whenever this constructor is called. To prevent code-clone, we encapsulate such block within a separate *MockMethod* in the *testClass* for reuse. Note that 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.

**Part 5—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 converts each statement in the code body following the new syntax based on Mockito.

The implementation of the above logic parts of the refactoring rely on the *ASTRewrite* mechanism of the *Eclipse JDT*.

## 3.3 Displaying the Refactoring Diff View

Once the refactoring is accomplished, the user can choose to *"Preview"* the generated refactoring solution. Here, *JMocker* will generate a side-by-side comparison of the candidate before the refactoring and after the refactoring as shown in Figure 4. *JMocker* follows the convention of the Git diff view—highlighting the added and removed lines of code in green and red backgrounds. This allows the user to inspect the refactoring solution, and implement as necessary improvements.

Figure 4 shows an example from *DataContextDataChannelEventsIT* in Cayenne, the left side shows the file before refactoring and the right side shows the file after refactoring. The testSubClass *MockChannelListener* is removed after refactoring, the testSubClass instance is replaced by a mock object created by Mockito. The assertion statements are replaced by *Mockito.verify()* to directly verify the method execution status and make test conditions clearer.

# 4 EVALUATION

We evaluated *JMocker* both quantitatively and qualitatively.

We evaluate *JMocker* on four open source projects, with a total of 610 test subclasses. They are: JackRabbit—an open source content repository for the Java platform [1], Log4J2—a Java-based logging utility [2], Qpid-Proton-J—a high-performance and lightweight messaging library [3] and Apache Commons—which focuses on all aspects of reusable Java Components, with 40 subprojects, including Commons-Collections, Commons-Lang, Commons-Logging, etc. Our evaluation focuses on different aspects of *JMocker*:

**Table 1: Evaluation Result**

| 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%) |

*Applicability.* *JMocker* is generally applicable to four different real-life projects. As shown in Table 1, JMocker detects 257 (column "#Candidates") refactoring candidates and successfully refactors 214 cases—indicating a 83% success rate over the feasible cases and 35% success rate over all 610 test subclasses. The remaining 43 (17%) cases associate with special syntax that is not captured in the filtering and/or refactoring process. 27 test sub-classes lead to compile errors (column "Comp.") due to syntax issues that were not

captured in the dataset of the empirical study. In addition, 16 test sub-classes, after the refactoring, lead to test behavior discrepancies (column "Discre.") due to special cases. One can keep refining our approach by incorporating these special cases.

*Test Behavior Preservation.* The test cases after the refactoring generally preserve test behaviors in terms of detecting potential defects in the production code. We inject 51811 mutations[8] to the production code, and investigate the execution status of each mutant—the coverage, survived or killed—before and after the refactoring. If the execution status of each mutant remains consistent, it suggests that the test case behaviors remain consistent in terms of detecting potential defects (i.e. mutants). The results show that $< 0.001\%$ of the generated mutants changed their coverage, and only 1% of the covered mutants changed survived/killed status. 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.

*Code Coupling.* We found that the refactoring overall decreases code complexity by removing 24% to 38% of the inheritance and 5% to 12% of the regular dependencies in the four evaluation projects.

*Efficiency.* The run-time performance of *JMocker* ranges from 30 to 250 seconds for detecting all refactoring candidates in a project, and on average, it takes 1 second to refactor a case.

*Feedback From Developers.* We conduct a user study involving six full-time developers to review the representative refactoring solutions generated by *JMocker*. The feedback shows: 1) The refactoring solutions generated by *JMocker* are of good quality. 2) The refactoring solutions generated by *JMocker* improve the cohesion and conciseness of test code, make test condition more explicit, and decouple test code from production code. And 3) the solution generated by *JMocker* can serve as an efficient first step in refactoring, developers can improve the test logic to further enhance unit tests.

## 5 DISCUSSION OF JMOCKER FOR EDUCATION

We also conducted a user study that leverages *JMocker* to facilitate the learning of Mockito for education. We invited five students majoring in Computer Science. We created a video to illustrate the auto generated refactoring solutions for three real-life cases based on the diff view. We provided the students both the video and the Mockito document as references to 1) implement refactoring for another real-life case and 2) use Mockito in three simple test cases for a toy project. Unfortunately, no student successfully finished the tasks. However, the comments from students suggest that Mockito is an advance testing technique poses challenges for beginners. One the one hand, the official document, containing overwhelming details, is not considered helpful in the process. On the other hand, our video, a highly appreciated format according to the students, does not provide sufficient detail to cover the complex syntax required by the tasks. This motivates us to think about better ways to facilitate the learning for students as a future research direction.

## 6 CONCLUSION

Despite the existance of powerful mocking frameworks, developers often turn to inheritance as a sub-optimal design for mocking. We presented an Eclipse plugin tool, named *JMocker*, which can automatically search for the usage of inheritance and replace it by Mockito for mocking. We evaluated our framework on *four* open-source projects and proved that *JMocker* is efficient, generally applicable to real-life projects, and preserves test behaviors. A user study involving real-life developers highlights the various benefits of *JMocker* to replace the usage of inheritance by Mockito. A training session that helps students to learn how to use Mockito based on the diff view of *JMocker* motivates us to think about better ways to teach this technique as a future research direction.

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] [n. d.]. https://jackrabbit.apache.org/jcr/index.html.
[2] [n. d.]. https://logging.apache.org/log4j/.
[3] [n. d.]. https://qpid.apache.org/.
[4] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103. https://doi.org/10.1109/FOSE.2007.25
[5] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 107–118.
[6] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. 2009. ReAssert: Suggesting repairs for broken unit tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 433–444.
[7] Jeff Grigg. 2012. http://wiki.c2.com/?ArrangeActAssert/.
[8] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.
[9] Matias Martinez, Anne Etien, Stéphane Ducasse, and Christopher Fuhrman. 2020. Rtj: a Java framework for detecting and refactoring rotten green test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 69–72.
[10] Gustavo Pereira and Andre Hora. 2020. Assessing Mock Classes: An Empirical Study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 453–463. https://doi.org/10.1109/ICSME46990.2020.00050
[11] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29. https://doi.org/10.1109/MS.2006.91
[12] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock? An empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 402–412. https://doi.org/10.1109/MSR.2017.61
[13] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing java systems. *Empirical Software Engineering* 24, 3 (2019), 1461–1498. https://doi.org/10.1007/s10664-018-9663-0
[14] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 540–552.