An Empirical Study on the Usage of Mocking Frameworks in Software Testing

Shaikh Mostafa, Xiaoyin Wang
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas, USA 78249
Email: {Shaikh.Mostafa, Xiaoyin.Wang}@utsa.edu

***Abstract*—In software testing, especially unit testing, it is very common that software testers need to test a class or a component without integration with some of its dependencies. Typical reasons for excluding dependencies in testing include the unavailability of some dependency due to concurrent software development and callbacks in frameworks, high cost of invoking some dependencies (e.g., slow network or database operations, commercial third-party web services), and the potential interference of bugs in the dependencies. In practice, mock objects have been used in software testing to simulate such missing dependencies, and a number of popular mocking frameworks (e.g., Mockito, EasyMock) have been developed for software testers to generate mock objects more conveniently. However, despite the wide usage of mocking frameworks in software practice, there have been very few academic studies to observe and understand the usage status of mocking frameworks, and the major issues software testers are facing when using such mocking frameworks. In this paper, we reported an empirical study on the usage of four most popular mock frameworks (Mockito, EasyMock, JMock, and JMockit) in 5,000 open source software projects from GitHub. The results of our study shows that mocking frameworks are used in a large portion (about 23%) of software projects that have test code. We also find that software testers typically create mocks for only part of the software dependencies, and there are more mocking of source code classes than library classes.**

I. Introduction

In modern software development, testing is a major approach to software quality assurance before a software application is released. Though the main purpose of testing is to reveal errors in the software application itself, in modern software practice, software applications can hardly be executed and tested without any dependencies such as libraries, third-party services, external data, etc.

To perform testing efficiently and effectively, one significant challenge is to handle various software dependencies properly. It should be noted that testing a software application with all of its dependencies is not always appropriate or even possible. The reasons are three folds. First of all, some software dependencies such as web services and databases are very slow when they are invoked. Thus involving such dependencies in testing will slow down the whole testing process, which may be fine for system testing, but not acceptable in unit testing and regression testing which are typically performed whenever a change is committed. Second, in unit testing, the testers should focus on the unit (e.g., functions, classes) to be tested. Involving some unstable dependencies may result in test interference due to bugs in the dependencies, and make it more difficult to identify and fix bugs inside the unit. Third, in large software projects, it is common that software developers develop different modules simultaneously and integrate the modules later. In such cases, it is probable that testing a certain module requires another module that has not been developed yet. Also, when developing frameworks that support call-backs, it is common that the dependencies invoked by the call-backs are not available at the testing phase (the call-backs are supposed to be defined by users of a framework).

In the testing practice of object-oriented software development process, to resolve the issues brought by software dependencies, mock objects are a typical solution. Mock objects can be used to replace real software dependencies by simulating the relevant features of software dependencies. Typically, methods of mock objects are designed to return some desired values (either a constant value for all inputs, or different values according to different inputs) when specific input values are given. Furthermore, to guarantee the correct interactions between the software component under test and dependencies, mock objects may check their interactions with the software component under test. For example, a mock email sender object of an email client may check whether the email-sending method is ever invoked by the email client and whether the arguments passed into the mock object are in correct format.

Recently, to facilitate developers in developing mock objects, a number of mocking frameworks have been developed. These mocking frameworks provide APIs for creating new mock objects, setting return values of methods in the mock objects, and checking interactions between the component under test and the mock objects. Along with the wide usage of mock objects and mocking frameworks, there are also many debates among software developers and testers on their usages. For example, some developers argue that mock objects should be used whenever a dependency is invoked because involving real dependencies may result in bug interference, but other developers do not agree because involving mock objects will also bring in a lot of overhead in test execution and maintenance. Also, developers debate on whether mocking frameworks should be used for all mock objects, and what features of mocking frameworks are most useful.

Despite the development of various mocking frameworks, and debates about usage of mock objects and mocking frameworks, there are very few academic empirical studies to observe and understand how mocking frameworks are used in software testing practice. In this paper, we present an empirical study of the usage of mocking frameworks in 5,000 open source software projects from GitHub[1]. Specifically, we studied the popularity of mocking frameworks in the testing of open source software projects and how developers of open-source software projects are using the frameworks. Our study finds that mocking frameworks and mock objects are used widely among open source Java software projects, but software testers usually mock only a small number of software dependencies. We also find that some unique features specific to one mocking framework are frequently used, and there are more mocking of source code classes than mocking of library classes.

The main contributions of this paper are as below.

- We identify the emergence of mocking frameworks in software testing practice, and the lack of existing academic study in this area.
- We carried out a large scale empirical study on more than 5,000 open source software projects from GitHub.
- We discover a number of interesting facts about the usage status of mocking frameworks, including their popularity, most frequently used features, and the characteristics of mocked objects.

The rest of this paper is organized as below. In Section II, we introduce some background knowledge about the difference and commonality of four most popular mocking frameworks. In Section IV, we present the design and results of our empirical study. Before we conclude in Section VI, we introduce some related research efforts.

II. Background

In this section, we first introduce some basic background knowledge on mocking objects. Then, we introduce the mechanism of mocking frameworks and some major mocking frameworks.

A. Mocking Objects

Mocking objects are used in software testing to simulate software dependencies so that the testing process can be accelerated and the testing scope can be limited to the component under test (instead of going beyond the interface of dependencies and invoke potential bugs relevant to dependencies). To simulate real dependencies, mock objects typically have the same interface as the objects they mimic. Therefore, the client object remains unaware of whether it is using a real object or a mock object.

The major difference between mock object and other test doubles (i.e., objects simulating software dependencies, such as test stubs, fake objects) is that, on top of simulating software dependencies, mock objects also contain assertions of their own. Mock objects will examine the context of each invocation of their methods, on whether a method is invoked, whether several methods are invoked in correct order, and whether arguments of the methods are passed in correctly.

For example, when testing an email client, software testers may not use the real email server which has slow response and random failures. By contrast, they may construct a fake email server object that returns whatever expected by the software testers. The expected return value can be either a simple success code for the testing of normal functioning of the email client, or a server error code for the testing of error handling of the email client. Such a fake email server helps to reduce testing time as well as to explore the handling of server errors that are very hard to invoke with a real server. However, when such a simplified email server is used, it is hard for testers to tell whether the email client has invoked the email server in a correct order (if multiple-step configuration of the server is required), or whether the client has passed correct arguments to the server.

Therefore, the software testers need a mock email server that not only simulate the real email server and returns success or error code, but also check whether the client interacts with it correctly and passes correct arguments. With such a mock email server, software testers can perform quick and controllable testing of the email client without losing the thorough exploration of the interface between the email client and the email server.

B. Mocking Frameworks

As mentioned in the previous section, mocking objects are desirable for simulating software dependencies in software testing. However, since mock objects need to perform various checking of the invocations, it is usually nontrivial to write such mock objects, and thus the benefit of using mock objects can be undermined by the effort spent on writing mock objects.

To solve this problem, in the past decade, people have developed various mocking frameworks to generate mock objects automatically. A code sample of using a mocking framework is presented below. With a mocking framework, software testers first create an empty mock object (Line x). Then, for the created mock object, software testers can specify a number of invocation for the object to check, both on the order of the invocations and the correctness of the arguments (Line x). After that, in the test code, testers run the component under test as usual but with created mock objects (Line x). During the execution, if the specified invocation order or arguments are not satisfied, the mock object will throw a test failure so that the software testers know that something wrong happens (Line x).

```
1: @Test
2: public void testEmailClient(){
3:       Server sv = EasyMock.
               CreateMock(Server.class);
4:       Client c = new Client(sv);
5:       Email em = new Email(...)
          //record
6:       EasyMock.expect(sv.send(em)).
               andReturn(0);
          //replay
7:       EasyMock.replay(sv);
8:       c.send(em)
9:       EasyMock.verify(sv)
10:}
```

For each major programming language, there have been a number of mocking frameworks developed, such as Mockito,

[1]GitHub: http://github.com

TABLE I. Basic Information of Subjects Used in Study

Info Type	Avg.	Med.	Max.	Min
# Java Files	153	18	10,141	1
LOC	25,305	1,833	3,094,265	0
# Test Files	16.4	0	3,335	0
# Developers	3.0	1	39	1
# Versions	11.4	1	1694	1

EasyMock, and JMock for Java, pmock, mox for Python, and NMock, Moq for C#. All of these mocking frameworks support the basic features of mocking including the creation of mock objects, the specification of invocation orders and arguments, and the verification of the specifications. The differences between these mocking frameworks are mainly on the design of APIs and supporting of advanced specifications such as checking the number of invocations of a certain method.

III. Design

In this section, we introduce the design of our empirical study on the usage of mocking frameworks among developers of open source software projects.

A. Research Questions

In this study, we design experiments to answer the following research questions. By answering these questions, we try to understand whether and how software developers and testers use mocking frameworks to handle dependencies in the testing of open source software projects.

- **RQ1:** How popular are mocking frameworks used in the testing of open source software projects? Are developers trying to mock most or all of dependencies?
- **RQ2:** What features of mocking frameworks are most frequently used in the testing of open source software projects?
- **RQ3:** What types of dependencies developers tend to mock?

B. Subjects

To perform our empirical study, we randomly downloaded 5,000 Java software projects from Github. We choose to focus on Java software projects because Java is one of the most widely used object-oriented programming language, and due to the popularity of Java in open source community, there are large number of open source Java software projects may serve as subjects.

The basic information of the 5,000 Java software projects are presented in Table I.

In Table I, we present five types of basic information of the collected software projects, which are in the Lines 2-6 of the table. Specifically, Line 2 presents the number of Java Files in the project. It should be noted that some of the software projects may be developed in multiple programming languages and may involve other types of source code files. In our study, we focus only on the Java part of such software projects. Line 3 presents the total lines of source code in the Java source files. Line 4 presents the number of test files in the software projects. Specifically, we consider a Java source file as a test file if it uses any library classes from JUnit or TestNG, which are the two major test frameworks for Java and the only test frameworks we notice the collected software projects are using. It should be noted that, in some software projects, a test class C may extend a library class in test frameworks, and other test files use only C but not any library classes in test frameworks. So in our study, we also consider the test files that indirectly use library classes in test frameworks by using classes like C. Line 5 and Line 6 presents the number of developers and the development period of the software projects in days.

Due the large number of subjects, we are not able to present the basic information of individual software projects. Instead, we present the statistical information of the software projects. Column 2-5 present the average, medium, maximum and minimum of all types of information, respectively.

From Table I, we can see that the chosen subjects vary largely in their sizes, testing efforts, number of developers, and history. The average size of the subjects are 25KLOC, and the largest subject has more than 3 million lines of code. Furthermore, more than half of the subjects have no test code (the medium of number of test files is 0), and the average number of test classes is 16.4.

C. Process

To answer the research questions listed in Section III-A, for each of the collected subject, we first identify all the Java source files and test files. Then, we parse these files to extract the code relevant to mocking frameworks.

In particular, we consider the four most popular Java mocking frameworks in our study: Mockito, EasyMock, JMock, and JMockit. PowerMock is another popular mocking framework, but it is actually an extension of Mockito and EasyMock frameworks, and typically software projects using PowerMock will use at least one of Mockito or EasyMock frameworks, so we do not consider PowerMock in our study.

From the relevant code, we extract the information that can help answer our research questions, such as the API method in mocking frameworks used, the dependency classes being mocked, etc.

IV. Empirical Study

In this section, we present the details and results of our empirical study. Specifically, we present the study to answer the research questions 1 to 3 in Section IV-A, Section IV-B, and Section IV-C, respectively. We summarize our findings in Section IV-D, and present the threats to validity in Section IV-E.

A. Popularity of Mocking Frameworks

To answer the first research question, we study the number of software projects using mocking frameworks among the 5,000 downloaded software projects. Specifically, among the 5,000 software projects, 2,046 has at least one test class (a class importing and use any class from JUnit or TestNg). Among the 2,046 projects, 459 projects (23%) uses at least more mocking framework.

TABLE II. POPULARITY OF MOCKING FRAMEWORKS

Mocking Framework	# Projects
Using Mockito	340
Using EasyMock	106
Using JMock	45
Using JMockit	7
Total	459
Multiple Mocking Frameworks	45

TABLE III. USAGE OF MOCKING FRAMEWORKS IN TEST CLASSES

	Avg.	Med.	Max.	Min
# Test Files using Mocking Frameworks	16.8	5	1037	1
Proportion of Test files using Mocking Frameworks	29.8%	20%	100%	0.3%

On top of the popularity of mocking frameworks in open source software projects, we are also interested in the market share of various popular mocking frameworks. Therefore, we studied the usage of different mocking frameworks and presented the results in Table II. From the table, we can see that, among the four major mocking frameworks, Mockito is the most widely used and is used in more than 70% software projects using mocking frameworks. EasyMock and JMock rank second and third, and are used in about 20% and 10% of software projects using frameworks, respectively. JMockit is the least widely used among the four. It should be noted that this observation may not be generalized because it is from 5,000 randomly selected Java projects from GitHub.

Furthermore, we find that 45 software projects use more than one mocking frameworks. We further studied the projects and find that the main reason for software developers to use multiple mocking frameworks is that, different software testers are more familiar with different mocking frameworks so that they use different mocking frameworks in the testing. Another reason is that the software project is moving from one framework to another.

We observe that mocking frameworks are used widely in software projects. However, the data above just reveals whether a software project involves mocking frameworks in any test class. It is still not known whether mocking frameworks are used in many test classes or only a small number of test classes. To answer this question, we further studied the number and proportion of test classes that use mocking frameworks and present the results in Table III

From the table, we can see that, in software projects that use mocking frameworks, on average, 16.8 test classes are using mocking frameworks, accounting for about 29.8% of all test classes. This observation shows that software testers are not using mock objects for all dependencies. Also, it is not the case that mock objects are used very rare and in only some special cases.

Moreover, we studied the number of mock objects in one test class and present the results in Table IV. From the table, we can see that most test classes (77.2%) have created 3 or less mock objects. Second, we try to understand, for test class that uses mocking frameworks, how many dependency classes are mocked and checked. Specifically, we deem a test class as dependency class if it is imported by the test class. The results are presented in Table V. The table shows that on average about 17% of dependency classes are mocked by the software testers. Thus, software testers seem to mock only a small portion of all dependency classes of a test class.

TABLE IV. NUMBER OF MOCK OBJECTS IN TEST CLASSES

# Mock Objects	# Test Files	Proportion
1	1270	45.3%
2	587	21.0%
3	333	11.9%
4	210	7.5%
5	135	4.8%
6	73	2.6%
7	49	1.7%
8	40	1.4%
9	22	0.8%
10+	82	2.9%

TABLE V. PROPORTION OF MOCKED CLASSES

	Avg.	Med.	Max	Min
# Mocked Classes	2.7	2	40	1
# Dependency Classes	17.0	14	215	1
Proportion of Mocked Clases	17.8%	14.3%	100%	0.9%

B. Usage of Mocking Frameworks

To answer the second research question, we try to study what APIs in mocking frameworks are most widely used. Since different mocking frameworks have different API sets, in this phase, we study only Mockito and EasyMock, which are the most and second most widely used mocking frameworks. The results are presented in Table VI and Table VII

From Table VI and Table VII, we have the following two main observations. First of all, verification of mock objects are extensively used. This fact shows that software testers do not use mocking frameworks simply as a convenient tool for generating normal test stubs or fake objects, but are using the core features of mock objects (i.e., specifying and verifying interactions between the component under test and the software dependency). Second, several advanced APIs in Mockito are widely used. Such APIs include "spy" which partially mock an existing class (i.e., use the mock object for specified method invocations, but the real dependency for other method invocations), "times" that specify how many times a certain method is invoked, etc.

C. Mocked Dependencies

To answer the third research question, we studied the classes that are mocked by software testers using mocking frameworks. Specifically, we try to find out whether software testers tend to mock library classes and what are the mostly

TABLE VI. MOST POPULAR TEN APIS IN MOCKITO

API	Frequency	Description
Mockito.verify	2249	verify tester's specification of the mock object
Mockito.mock	1678	create an empty mock object
Mockito.when	1471	specify a method in the mock object to be invoked
Mockito.spy	403	partially mock an existing class
Mockito.times	232	specify how many times a method is invoked
Mockito.given	161	Alias of "when" for behavior driven development style
Mockito.never	152	specify that a method is never invoked
Mockito.verifyNoMoreInteractions	98	specify that there are no more interactions between the component under testing and the mock object
Mockito.doReturn	85	specify the return value of a method invocation
Mockito.verifyZeroInteractions	74	specify that there are no interactions between the component under testing and the mock object

TABLE VII. MOST POPULAR TEN APIS IN EASYMOCK

API	frequency	description
EasyMock.expect	671	specify a method in the mock object to be invoked
EasyMock.createmock	647	create an empty mock object
EasyMock.replay	638	notice the mock object that the testing of component under testing starts
EasyMock.verify	510	verify tester's specification of the mock object
EasyMock.expectLastCall	144	alias for "expect" for void methods
EasyMock.eq	90	expect an value that is equals with a given value
EasyMock.createNiceMock	67	create a mock object that check the order of method invocations
EasyMock.anyObject	64	match with any type of arguments when specifying the invocation of a method
EasyMock.reportMatcher	58	specify a matcher of arguments for specification of method invocations
EasyMock.capture	53	capture a matched value for later access

TABLE VIII. MOCKING OF LIBRARY CLASSES

	Avg.	Med.	Max	Min
# Mocked Library Classes	0.846211552888	0.0	18	0
Proportion of Library Classes	0.394204219345	0.0	1.0	0.0

mocked library classes. To find the answer, we check whether the mocked classes are library classes (i.e., classes that are not in the source code base of the software project), and present the results in Table VIII.The results show that about 39% of all mocked classes are library classes, so software testers actually tend to mock more classes in the source code, compared with library classes. This shows that the major reason for software testers to perform mocking may be reducing the

In Table IX, we present the top 10 mostly mocked library classes. Specifically, the frequencies are the times that the specific class is mocked in all the studied open source software projects. From the table, we can observe that the two major source of mostly mocked classes are the package "javax.servlet", and the package "javax.jcr". Specifically, the former package is about HTTP request and responses, and the latter package is about the operation of Java Content Repositories.

D. Summary

To sum up, the major findings of our empirical study are as below.

- Mocking frameworks and mock objects are used widely among open source Java software projects. However, software testers usually mock only a small number and portion of software dependencies.
- Verification is widely used to check the specified interaction between the component under testing and the mocked object. Special APIs in both Mockito and EasyMock are widely used, which implies an incompleteness of features for a single mocking framework.
- Software testers tend to mock source code classes than libraries, while library classes also take a substantial proportion (40%) in all mocked classes. The mostly mocked library classes are classes from packages handling HTTP requests / responses, and content repositories.

TABLE IX. MOSTLY MOCKED LIBRARY CLASSES

Mocked Library Class	Frequency
javax.servlet.http.HttpServletRequest	44
org.fest.assertions.description.Description	41
javax.servlet.ServletContext	33
javax.jcr.Node	32
rx.Observer	24
org.openqa.selenium.WebDriver	23
javax.servlet.http.HttpServletResponse	22
javax.jcr.Property	20
javax.jcr.Session	20
java.io.InputStream	17

E. Threats to Validity

The major threat to the internal validity of our study is the potential errors in the programs analyzing the collected software projects and performing statistics. To reduce the threat, we tried our best to write the code carefully to avoid any bugs in the programs. The major threat to the external validity of our study is that our observations may be specific to the subject software projects, and cannot be generalized. To reduce this threat, we use a large number of subject software projects, and choose the subjects randomly. Also, it is possible that our findings are specific to Java software projects. To further reduce this threat, we plan to carry out similar empirical studies on subjects written with other major programming languages in the future.

V. RELATED

As far as we know, this paper is the first research effort to study the usage status of mocking frameworks and mock objects in software practice. There have been a number of existing research efforts on enhancing mocking frameworks or leveraging mock objects to improve automatic software testing. Freeman et al. [1] [2] presented the basic process and concepts of using mocks objects in unit testing, as well as a mocking framework jMock. Galler et al. [3] proposed an approach to automatically generation of mocking objects satisfying certain preconditions to serve as test inputs in automatic test-case generation for Java unit testing. Taneja et al. [7] proposed to automatically generate mock objects to simulate the behavior of database systems in the automatic test-case generation of database systems. Coelho et al. [8] proposed an approach to generate mock agents in the unit testing of multi-agent software systems. Due to the necessity of mock objects in unit testing, researchers also proposed approaches to automatically generating mock objects along with unit test cases [9] [10]. For the existing test cases which are not using mock objects, Saff and Ernst [11] proposed an approach to automatically refactor such test cases by adding mock objects. Marri et al. [12] carried out an experiment to study the benefit of using mock objects to simulate file systems. All these research efforts are about automatic generation of mock objects and how to leverage mock objects in specific testing problems, while our work focuses on the current usage status of mocking frameworks and mock objects in real world software projects.

There are also some existing studies investigating the status of testing practice, or the effectiveness of testing techniques on real-world open source projects. Singh et al. [13] empirically explored 20,000 open-source projects, and found that bigger projects have a higher probability to contain test cases and projects with more developers may have higher number of test cases. In addition, they also found that the number of test cases has a weak correlation with the number of bugs. Greiler et al. [4] explored the current testing practices currently used for the specific plug-in systems. Pham et al. [5] investigated how social coding sites influence testing behavior. They found several strategies that software developers and managers can use to positively influence the testing behavior. Fraser et al. [6] empirically evaluated automated test generation on 100 real-world Java projects. They found that high coverage is achievable on commonly used types of classes, and also identified future directions for further improve code coverage. In this paper, we aims to investigate the differences between manual tests and DSE-based tests for real-world systems.

VI. Conclusion

In this paper, we direct a large scale empirical study on the usage of mocking frameworks in software testing. To perform the study, we collected 6,000 Java software projects from Github, and analyze the source code of the projects to answer a number of questions about the popularity of mocking frameworks, the usage of mocking frameworks, and the mocked classes. Our major findings include that mocking frameworks are widely used in practice, and a small portion of dependencies are mocked. This finding shows the requirement of more research on mocking frameworks, as well as on the reasons why developers choose to mock a class while not to mock the other. Furthermore, we find that a number of unique features in Mockito and EasyMock are widely used. This implies that it is possible to build better mocking frameworks by incorporating the most popular features of existing mocking frameworks. We also find that software developers tend to do more mocks on source code classes than library classes, which is not as we expected.

In the future, we plan to extend our work in the following directions.

First of all, we plan to direct similar studies on software projects written in programming languages other than Java to check whether our findings can be generalized.

Second, we plan to develop techniques to help software developers and testers make decisions on whether or not a class should be mocked.

Third, we plan to develop techniques to generate mock objects automatically with mocking frameworks in automatic unit testing.

References

[1] Freeman, Steve and Mackinnon, Tim and Pryce, Nat and Walnes, Joe, Mock Roles, Objects, Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, 236–246, 2004.

[2] Freeman, Steve and Mackinnon, Tim and Pryce, Nat and Walnes, Joe, jMock: Supporting Responsibility-based Design with Mock Objects, Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications,4–5, 2004.

[3] Galler, Stefan J. and Maller, Andreas and Wotawa, Franz, Automatically Extracting Mock Object Behavior from Design by Contract & Trade Specification for Test Data Generation Proceedings of the 5th Workshop on Automation of Software Test, 43–50, 2010.

[4] M. Greiler, A. van Deursen, and M. Storey. Test confessions: a study of testing practices for plug-in systems. In Software Engineering (ICSE), 2012 34th International Conference on, pages 244–254, 2012.

[5] R. Pham, L. Singer, O. Liskin, K. Schneider, et al. Creating a shared understanding of testing culture on a social coding site. In Software Engineering (ICSE), 2013 35th International Conference on, pages 112–121, 2013.

[6] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In Proceedings of the 2012 International Conference on Software Engineering, pages 178–188, 2012.

[7] Taneja, Kunal and Zhang, Yi and Xie, Tao, MODA: Automated Test Generation for Database Applications via Mock Objects, Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 289-292, 2010.

[8] Coelho, Roberta and Kulesza, Uirá and von Staa, Arndt and Lucena, Carlos, Unit Testing in Multi-agent Systems Using Mock Agents and Aspects, Proceedings of the International Workshop on Software Engineering for Large-scale Multi-agent Systems, 83-90, 2006

[9] Islam, Mainul and Csallner, Christoph, Dsc+Mock: A Test Case + Mock Class Generator in Support of Coding Against Interfaces, Proceedings of the Eighth International Workshop on Dynamic Analysis, 26–31, 2010.

[10] Pasternak, Benny and Tyszberowicz, Shmuel and Yehudai, Amiram, GenUTest: A Unit Test and Mock Aspect Generation Tool, Proceedings of the 3rd International Haifa Verification Conference on Hardware and Software: Verification and Testing, 252–266, 2008.

[11] Saff, David and Ernst, Michael D. Mock Object Creation for Test Factoring, Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 49–51, 2004.

[12] Marri, M.R.; Tao Xie; Tillmann, N.; De Halleux, J.; Schulte, W., An empirical study of testing file-system-dependent software with mock objects, ICSE Workshop on Automation of Software Test, 149–153, 2009.

[13] K. P. Singh, T. F. Bissyandé, D. Lo, L. Jiang, et al., An empirical study of adoption of software testing in open source projects. In Proceedings of the 13th International Conference on Quality Software (QSIC 2013), pages 1–10, 2013.