Indispensable Mocks

Mohammad Mahdi Abdollahpour
Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
mohammadmahdi.abdollahpour@uwaterloo.ca

***Abstract*—More than two decades ago, Mock Object was introduced as an extension to Test-Driven Development with the primary objective of addressing the challenges faced by developers when isolating the system under test during unit testing. The idea of using mock objects has been widely adopted and numerous frameworks and libraries have been developed based on this concept for various programming languages. However, there is limited academic research on this topic. This paper aims to encourage practitioners to increase their focus on this technique by introducing seven predicates that require the use of a mocking framework. We substantiate our predicates by providing explanatory justifications and practical examples. Additionally, we conduct an empirical analysis on 30 Java projects and identify mock usages that are not easily discernible based on our predicates. Finally, we offer our interpretation of the findings to assist and inspire future researchers to continue this work.**

***Index Terms*—mock objects, unit tests, program analysis, static analysis, empirical studies**

I. Introduction

Extreme Programming (XP) [1] advocates for unit testing as one of its fundamental practices toward Test-Driven Development (TDD). It distinguishes "unit tests" from "functional tests" via the differences in perspectives: programmer vs. customer. [2] introduces the concept of "mock objects" as a technique for tackling parts of the challenges in the practical implementation of unit testing. [3] gives a concise definition of mock objects (based on [4]): "objects pre-programmed with expectations which form a specification of the calls they are expected to receive". There is a close semantic relation between mocks and another form of test double called "stubs" which was first introduced by [5]. The main difference between them lies in the way they verify the expectations: stubs verify states while mocks verify behavior [3]. [4] uses an umbrella term for all forms of replacements for real objects: Test Doubles.

Aside from terminology, there are benefits and drawbacks to using mocks. [2] makes a case for using mocks based on the following benefits:

- Deferring infrastructure choices such as database implementation
- Setting up complex system state in a localized manner
- Coping with test conditions that are difficult to reproduce
- Shorter feedback loop due to faster and localized failures
- Better understanding of the system through unexpected mock failures
- Reducing the need to expose the structure of domain code
- Encouraging object passing over singletons which decreases the number of unexpected side-effects
- Encouraging smaller and more specialized classes
- Interface discovery

Years later, in [6], the same authors introduce Need-Driven Development based on the most important benefit of using mocks: interface discovery. [6] considers testing as a design activity and states that TDD with mock objects allows developers to design an object by its requirements rather than what it provides.

Originally, using mock objects was introduced as a technique; however, many tools and libraries have been developed to ease the process of implementing it. The most notable libraries for Java include JMock [6], EasyMock [7], and Mockito [8]. These libraries provide generalized APIs to create mock objects by implementing a domain-specific embedded language within Java.

Despite the aforementioned benefits, the use of mock objects has been met with significant opposition among practitioners. For instance, Fowler has expressed concern about coupling tests to implementation in [3], while Martin advocates against using mocking tools due to the learning curve and the use of a new domain-specific language [9].

In this research, our focus is on the necessity of using mock objects. Specifically, we aim to answer the following research questions:

- RQ1: Under what circumstances is it necessary to use mock objects, i.e., when are the alternative forms of test doubles impractical?
- RQ2: To what extent is the utilization of mocking considered necessary in Java projects?

We posit that there are situations where not using mocking tools is highly impractical. To this end, we provide detailed explanations and concrete examples of real-world scenarios that necessitate using mock objects.

The contributions of this study are as follows:

- A list of rules and examples that outline when using mock objects is necessary
- An empirical analysis of mock usage in several open-source Java projects

II. Related Work

In the field of software engineering and software testing, there has been an ongoing debate regarding the use of terminology. As documented by [4], there exist differences in how different groups of practitioners define terms such as "unit testing" and "integration testing". For instance, some define unit testing in contrast to integration testing, focusing on scope rather than perspective. Fowler [3] writes, "... you are focusing on one element of the software at a time -hence the common term unit testing.".

Rainsberger [10] has expressed a strong stance against the use of integration tests, advocating instead for a tree-like architecture where every interaction between layers of dependency is verified by four types of tests, two of which require mock objects.

The use of mock objects has also generated controversy, with authors such as Langer [11], Fowler [3], and Beck [12] arguing against their use except when absolutely necessary. This viewpoint is known as "Classicist" or "Chicago" (as opposed to "Mockist" or "London"). The classicist position cites issues such as "tautological tests" [13] and "drifting test doubles" [14] to argue against the use of mocks.

Several research studies have been conducted in the area of mock objects, exploring different aspects of their design, implementation, and usage. For instance, Kim et al. [15] proposed a formalization of existing mock object models and proposed a new one. Additionally, Saff and Ernst [16] introduced the concept of test factoring with mock objects, while Tillmann and Schulte [17] developed a .NET-based tool for generating mock objects and their corresponding behavior through symbolic execution. Similarly, Pasternak et al. [18] created a Java-based tool that automatically generates unit tests and mock aspects for isolation testing. In another work, Galler et al. [19] presented an approach to derive mock object behavior from Design by Contract^{TM} specifications. Furthermore, [20] proposes a technique for generating mock classes and test cases for interface-coded objects. Lastly, Taneja et al. [21] leverages dynamic symbolic execution to generate tests for a database application using mock objects.

III. Defining Indispensable Mocks

The utilization of mock objects presents the possibility of producing advantageous [2][22] outcomes, as well as potentially damaging ones [11]. Through an examination of relevant literature and a manual investigation of open-source projects written in Java, we have arrived at the conclusion that certain circumstances encountered during unit testing require the use of mock objects. In this section, we concentrate on outlining the conditions in which eschewing mocks is impractical. To this end, we provide a list of structural and semantic predicates, accompanied by their corresponding justifications and real-world usage examples. If any predicate evaluates to true on a type *T*, which is mocked, we classify *T* as indispensable.

A. P0: Computational expense

Predicate. *T* is a class or abstract class or interface that has a method (or a public constructor) *M* which needs to be invoked in a unit test while it is computationally expensive to execute any "meaningful" implementation of *M*.

Justification. The concept of fast-running tests has been emphasized in several sources, including [23][4][24]. A key reason for the widespread utilization of mock objects is their ability to address the practical reality that employing a real implementation can require an excessive amount of time to execute. While in certain scenarios, it may be feasible to use fake data or simplified alternatives, these solutions may not adequately address the need to verify the behavior of the system in most cases (i.e., limited to state verification).

Example.

```
HttpClient mockHttpClient = mock(HttpClient.class);
HttpResponse<String> mockResponse =
    mock(HttpResponse.class);
when(mockHttpClient.send(
      HttpRequest.newBuilder()
         .uri(URI.create(url))
         .GET()
         .build(),
      HttpResponse.BodyHandlers.ofString()))
   .thenReturn(mockResponse);
when(mockResponse.statusCode()).thenReturn(200);
when(mockResponse.body()).thenReturn("{\"temperature\":
    20, \"humidity\": 80}");
WeatherService weatherService = new
    WeatherService(mockHttpClient);
Weather weather =
    weatherService.getWeatherByCity("Waterloo");
assertEquals(20, weather.getTemperature());
assertEquals(80, weather.getHumidity());
```

In this instance, Mockito [8] is employed to generate mock objects for the `HttpClient` and `HttpResponse` classes. The mock objects are configured to imitate the behavior of a real web service but produce fake data when queried.

Utilizing mock objects in this manner enables us to test our `WeatherService` class without making actual calls to the external web service, which can prove to be both time-consuming and costly.

B. P1: Final class

Predicate. *T* is a final class with a method *M* which needs to be invoked in a unit test.

Justification. Sub-classing of final classes is not allowed in Java (which is the point of using final class in the first place), thus it is not possible to verify the behavior of an object that calls a method of an object of a final class during the test time. If the developer attempts to create a stub for a final class, they will fail to inherit from the final class in order to add the test behavior to it.

Example.

```
@Mock
private PaymentGateway paymentGateway;
private PaymentService paymentService;
@Before
public void setUp() {
   MockitoAnnotations.initMocks(this);
   paymentService = new PaymentService(paymentGateway);
}
@Test
public void testProcessPayment() {
   when(paymentGateway.processPayment(anyDouble(),
        anyString())).thenReturn(true);
```

```
    boolean result = paymentService.processPayment(100.0,
        "123456789");
    verify(paymentGateway).processPayment(100.0,
        "123456789");
    assertTrue(result);
  }
```

It would not be possible to write the test for `PaymentService` class that depends on a final `PaymentGateway` class without using a mocking framework. This is because a final class cannot be extended or sub-classed. This limitation makes it impossible to create a test double of the final `PaymentGateway` class.

Without using a mocking framework, we would have to use the real `PaymentGateway` class in our tests. This would make our tests more complex and less reliable, as they would depend on external factors such as network connectivity, database availability, or API stability. In addition, testing with the real `PaymentGateway` class could have financial consequences, especially if the payment processing is not done in test mode.

By using a mocking framework like Mockito, we can create a mock object of the final `PaymentGateway` class and simulate its behavior in our tests. This allows us to isolate the behavior of the `PaymentService` class and test it in a controlled environment without relying on external factors.

C. P2: Final method

Predicate. T is a class or abstract class with a final method M which needs to be invoked in a unit test.

Justification. This issue is similar to the problem of final classes but in a smaller scope. There is no easy way for a developer to override a final method with a custom behavior specifically designed for unit testing.

Example.

```
public class PaymentProcessor {
  // ...
  public final boolean processPayment(
    String paymentDetails) { ... }
}
```

```
@Test
public void testProcessOrder() {
  PaymentProcessor mockPaymentProcessor =
      mock(PaymentProcessor.class);
  when(mockPaymentProcessor.processPayment(anyString()))
    .thenReturn(true);
  OrderProcessor orderProcessor = new
      OrderProcessor(mockPaymentProcessor);
  boolean result =
      orderProcessor.processOrder("paymentDetails");
  assertTrue(result);
  verify(mockPaymentProcessor,
      times(1)).processPayment("paymentDetails");
}
```

In the absence of a mocking framework, testing the class that relies on the final method would prove to be challenging since we are unable to modify the behavior of the final method during testing. As a result, we would need to execute the final method in the test environment, which could potentially be time-consuming, undependable, and may impede the stability of the system.

It is worth noting that stubbing final methods and final classes is a difficult endeavor in Java and is not available in several mocking frameworks.

D. P3: Surfeit of methods

Predicate. T is an interface that defines at least N methods. (N is a number greater than 1 and is chosen by the designer of the system.)

Justification. The creation of a custom class that implements a specific interface may appear to be a straightforward task. Nevertheless, if the number of methods within the interface exceeds a certain threshold, it becomes burdensome (both in terms of development and maintenance) to write an implementation for each individual method of the interface.

Example.

```
interface PaymentGateway {
  void authenticate(String apiKey);
  boolean authorize(String ccNumber, double amount);
  void capture(String transactionId, double amount);
  void refund(String transactionId, double amount);
  void voidTransaction(String transactionId);
  String getTransactionStatus(String transactionId);
  void setShippingAddress(String address);
  void setBillingAddress(String address);
  void setCustomerName(String name);
}
```

```
@Test
public void testPaymentService() {
  when(paymentGateway.authorize(anyString(),
      anyDouble())).thenReturn(true);
  doNothing().when(paymentGateway).capture(anyString(),
      anyDouble());
  doNothing().when(paymentGateway).refund(anyString(),
      anyDouble());
  doNothing().when(paymentGateway)
    .voidTransaction(anyString());
  when(paymentGateway.getTransactionStatus(anyString()))
    .thenReturn("success");
  boolean result =
    paymentService.processPayment("12345", 100.00);
  assertTrue(result);
  verify(paymentGateway, times(1)).authorize(anyString(),
      anyDouble());
  verify(paymentGateway, times(1)).capture(anyString(),
      anyDouble());
  verify(paymentGateway, never()).refund(anyString(),
      anyDouble());
  verify(paymentGateway,
      never()).voidTransaction(anyString());
  verify(paymentGateway,
      times(1)).getTransactionStatus(anyString());
}
```

It can be challenging to implement a comprehensive test for an interface with a lot of methods (`PaymentGateway` in our example) without using a mocking framework. Without a mocking framework, we would need to create a complete implementation of the interface, which can be time-consuming and difficult to maintain. We would also need to ensure that the implementation provides consistent and predictable behavior for all methods in all possible scenarios.

Furthermore, if the implementation of the interface is complex, it may be difficult to set up the required dependencies and test environment for a thorough test. This can be especially true for testing scenarios that involve multiple objects interacting with each other.

E. P4: No public constructor

Predicate. T is a class or abstract class without any public constructor.

Justification. The instantiation of a class with no public constructor is not a trivial task. In some cases, factory classes are employed to exert greater control over the instantiation process. However, during testing, creating the entire setup solely to instantiate a single object may not be considered a feasible option.

Example.

```java
public class PaymentGatewayFactory {
   private static PaymentGatewayFactory instance = new
        PaymentGatewayFactory();
   private PaymentGatewayFactory() {
      // private ctor...
   }
   public static PaymentGatewayFactory getInstance() {
      return instance;
   }
   public PaymentGateway createPaymentGateway() {
      PaymentGateway paymentGateway = new
           PaymentGateway();
      // set up the PaymentGateway object...
      return paymentGateway;
   }
}
```

```java
@Test
public void testProcessPayment() {
   PaymentGateway mockPaymentGateway =
        Mockito.mock(PaymentGateway.class);
   PaymentProcessor paymentProcessor = new
        PaymentProcessor(mockPaymentGateway);
   when(mockPaymentGateway.processPayment(
     any(PaymentRequest.class)))
      .thenReturn(new PaymentResponse("success"));
   PaymentRequest paymentRequest = new
        PaymentRequest("123", 100.0);
   PaymentResponse paymentResponse =
        paymentProcessor.processPayment(paymentRequest);
   verify(mockPaymentGateway)
     .processPayment(paymentRequest);
   assertEquals(paymentResponse.getStatus(), "success");
}
```

In this example, the `PaymentProcessor` class is responsible for processing payments in a banking application. The `PaymentProcessor` class relies on the `PaymentGateway` class, which facilitates communication with a third-party payment service. The `PaymentGateway` class has a private constructor and is instantiated via a factory class called `PaymentGatewayFactory`. To test the `PaymentProcessor` class, a mock `PaymentGateway` object can be employed in place of an actual one.

Testing the `PaymentProcessor` class without the use of mocks would necessitate writing code to configure the factory class and ensuring that all of its dependencies and context are correctly established. A real implementation of `createPaymentGateway` would potentially entail expensive operations that must be executed in every test. Moreover, adding test behavior to the `PaymentGateway` class would not be possible because sub-classing it would also be infeasible due to the absence of a non-private constructor. Under these circumstances, verifying the behavior of the `PaymentProcessor` would be a challenging task.

F. P5: Complex public constructor

Predicate. T is a class or abstract class and all of its public constructors are complex. A public constructor is complex if it accepts more than N arguments or accepts at least one non-nullable argument; type of which matches at least one of the predicates. (N is a number greater than 1 and is chosen by the designer of the system.)

Justification. In Java, classes are usually instantiated through their public constructors. Nevertheless, when the only viable option is to utilize a public constructor that requires a significant number of arguments, each of which could potentially necessitate separate instantiation along with their own setup and dependencies, it becomes exceedingly difficult to write a stub for such a class.

Example. Imagine we have a `PaymentService` class with the following constructor.

```java
public PaymentService(PaymentGateway paymentGateway,
     AccountRepository accountRepository,
     FraudDetectionService fraudDetectionService,
     NotificationService notificationService, Logger
     logger, UserService userService,
     CurrencyConversionService currencyConversionService)
{
   this.paymentGateway = paymentGateway;
   this.accountRepository = accountRepository;
   this.fraudDetectionService = fraudDetectionService;
   this.notificationService = notificationService;
   this.logger = logger;
   this.userService = userService;
   this.currencyConversionService =
        currencyConversionService;
}
```

Then we can have a test like:

```java
@Mock
private PaymentService paymentServiceMock;
@Test
public void shouldProcessPaymentSuccessfully() {
   PaymentProcessor paymentProcessor = new
        PaymentProcessor(paymentServiceMock);
   Payment payment = new Payment("John Smith",
        "1234-5678-9012-3456", 100.00);
   when(paymentServiceMock.processPayment(payment))
     .thenReturn(true);
   boolean result =
        paymentProcessor.processPayment(payment);
   verify(paymentServiceMock,
        times(1)).processPayment(payment);
```

```
    assertTrue(result);
}
```

In the absence of mock frameworks, it becomes necessary to manually create and instantiate all the dependencies of the `PaymentService` class. This can be an arduous and error-prone process, particularly when dealing with dependencies that are complex and have their own dependencies. Additionally, creating and managing all the dependencies in a testing environment can make the tests more complicated and harder to read and maintain.

G. P6: Complex method

Predicate. *T* is a class or abstract class or interface with a complex method *M* that needs to be invoked in a unit test. A method is complex if its return type matches at least one of the predicates.

Justification. When attempting to include test behavior in a method, a developer may endeavor to create a customized, simplified version of the method for a stubbed class. However, when the instantiation of the return value poses technical difficulties, it becomes challenging to develop and maintain a straightforward stub implementation for the purpose of unit testing.

Example.

```
@Test
public void testProcessPayment() {
    PaymentService paymentService =
         mock(PaymentService.class);
    User user = mock(User.class);
    PaymentDetails paymentDetails =
         mock(PaymentDetails.class);
    Account account = mock(Account.class);
    BillingAddress billingAddress =
         mock(BillingAddress.class);
    CreditCard creditCard = mock(CreditCard.class);
    PayPalAccount payPalAccount =
         mock(PayPalAccount.class);
    Invoice invoice = mock(Invoice.class);
    PromotionCode promotionCode =
         mock(PromotionCode.class);

    PaymentProcessor paymentProcessor = new
         PaymentProcessor(paymentService);

    paymentProcessor.processPayment(user, paymentDetails,
         account, billingAddress, creditCard,
         payPalAccount, invoice, promotionCode);

    verify(paymentService).processPayment(same(user),
         same(paymentDetails), same(account),
         same(billingAddress), same(creditCard),
         same(payPalAccount), same(invoice),
         same(promotionCode));
}
```

Manually creating all the dependencies in the code would lead to verbosity and decreased readability. It would also require a considerable amount of setup code, making the test difficult to maintain. Additionally, this approach could result in slow and brittle tests that might break if any of the dependencies change, making it hard to isolate and test the `PaymentProcessor` class.

IV. Finding Indispensable Mocks

To answer the second research question, we conduct an analysis to find matches for our predicates. For the scope of this study we focus on implementing predicates *P1* to *P4* along with a simplified version of predicate *P5*. More specifically, we limit the implementation of predicate *P5* to the argument counts, thus we do not recursively run over the predicates.

Approach. First, we implement a tool using a declarative approach on top of Doop [25]. We define Datalog rules to find mock objects (based on the syntax of Mockito [8] library). We also add rules to find method invocations on the mock objects. Our rules allow us to track mockness through fields, collections, and arrays. We provide a selected subset of the Datalog rules.

```
// v = mock()
isMockVar(v) :-
  AssignReturnValue(mi, v),
  callsMockSource(mi).
// v = (type) from
isMockVar(v) :- isMockVar(from),
  AssignCast(_/* type */, from, v, _/* inmethod */).
// v = v1
isMockVar(v) :- isMockVar(v1),
  AssignLocal(v1, v, _).
```

Listing 1: Selected rules for Datalog mock analysis

```
// v = callee(), where callee's return var is mock
isInterprocMockVar(v) :-
  AssignReturnValue(mi, v),
  mainAnalysis.CallGraphEdge(_, mi, _, callee),
  ReturnVar(v_callee, callee),
  isMockVar(v_callee).

// callee(v) results in formal
//  param of callee being mock
isInterprocMockVar(v_callee) :- isMockVar(v),
  ActualParam(n, mi, v),
  FormalParam(n, callee, v_callee),
  mainAnalysis.CallGraphEdge(_, mi, _, callee),
  Method_DeclaringType(callee, callee_class),
  ApplicationClass(callee_class).
```

Listing 2: Two rules give interprocedural analysis in Doop.

```
// v = c[idx]
isMockVar(v) :-
  isArrayLocalThatContainsMocks(c),
  LoadArrayIndex(c, v, _ /* idx */).

// c[idx] = mv
isArrayLocalThatContainsMocks(c) :-
  StoreArrayIndex(mv, c, _ /* idx */),
  isMockVar(mv).
```

Listing 3: Rules for handling arrays.

After finding mock objects, we post-process the generated Doop results to answer our research question. The post-processing is done in two steps. First, we directly parse and analyze selected Doop fact and result files to collect the following information about all mock objects:

- The types of the mocked variables
- Exact location of the mock object (surrounding class and method name)
- The list of all final method invocations of each of the mocked types
- Class modifiers (public, private, final, ...)
- The count of methods in each of the mocked types
- List of public constructors' signatures for each of the mocked types

In the next step, we implement and run each of the predicates over all the data collected from the previous step. We use $N = 5$ in our implementation of the predicates *P3* and *P5*.

Result. The results are shown in table I.

TABLE I: Total number of mocked types in the benchmark projects along with the number of mocked types matching each of the predicates

Benchmark	# Mocked Types	P1	P2	P3	P4	P5
vraptor-parent	84	0	0	14	0	0
lettuce-core	43	0	1	14	3	1
datasource-proxy	42	0	1	12	1	2
openshift-restclient-java	38	0	0	16	2	0
mybatis-3-mybatis	24	0	0	13	4	0
jsonschema2pojo	22	1	0	3	11	0
maven-assembly-plugin	18	0	0	8	0	1
bootique	15	0	0	2	3	0
thumbnailator	14	0	0	0	2	2
zip4j	11	0	0	3	2	0
palatable_lambda	11	0	0	1	7	0
perwendel_spark	10	0	0	4	2	0
flink	7	0	0	1	0	0
java-faker	7	0	0	0	2	0
braintree_java	6	0	0	0	0	0
maven-dependency-plugin	6	0	0	3	1	1
minimal-json	6	0	0	0	3	0
quartz	5	0	0	4	0	0
offheap-store	3	0	0	0	0	0
JSON-java	3	0	0	1	0	0
mbassador	3	0	0	0	0	0
exec-maven-plugin	3	0	0	1	0	1
signpost	3	0	0	2	0	0
javapoet	3	0	0	2	0	0
commons-collections	3	0	0	1	0	0
azure-functions-java-worker	3	3	0	0	3	0
plexus-resources	2	0	0	1	0	0
java-gitlab-api	2	0	0	0	1	0
RxRelay	2	0	0	0	0	0
jdeb	2	0	0	0	0	0

Analysis. The observation is made that the practice of mocking final classes and final methods is exceedingly uncommon. There exist two possible explanations for this phenomenon. Firstly, the employment of final classes and methods may not be widespread in Java (though further research is needed to verify this claim). Secondly, the inability to mock final methods and final classes was pervasive in most mocking frameworks for an extended period. The tendency to abstain from mocking final classes or the lack of awareness concerning newer versions of mocking frameworks may have contributed to this occurrence.

On the contrary, we can observe that mocking interfaces with an excessive number of methods is highly prevalent among developers. We believe that the chief reasons for this are to evade code clutter and to tackle the challenge of maintenance. Additionally, we see that mocking classes with no public constructors is comparatively frequent. Factory and builder patterns are extensively employed in Java, and they typically wrap around a class with private constructors. In our view, the widespread use of these two patterns and the inability to subclass classes with only private constructors are among the reasons that have given rise to this practice.

Finally, we note that developers do not commonly utilize mock objects for public constructors with an excessive number of arguments. This may be attributed to the fact that such classes are not commonly encountered due to the general consensus among Java developers that having an excessive number of arguments in a public constructor (or any method) is not advisable, as per the principle outlined in [23].

V. Threats to Validity

Construct Validity. While we acknowledge that the predicates proposed in this paper are not exhaustive, we contend that compiling a complete inventory of rules that account for every conceivable scenario involving the use of mock objects is unfeasible. Nevertheless, we have undertaken a manual analysis of over 30 Java projects, along with their corresponding usage of mocking frameworks, to identify trends, patterns, and plausible justifications for incorporating the mocking technique.

Internal Validity. Based on the data presented in table I, we introduced a few plausible rationales for the patterns we have observed. We concede that most of these conjectures are not fully backed by formal research. Nevertheless, we do not purport that these arguments are established empirical truths, but rather represent our interpretations derived from the systematic examination of the benchmark projects and our general familiarity with the Java community. Furthermore, we believe that several of the inferences we have made could potentially serve as the focus of future research endeavors

External Validity. Our study is limited to Java benchmarks, and thus, the proposed predicates and the obtained results may not be generalizable to other programming languages. We recognize this as a potential shortcoming of our investigation, and therefore, we endeavor to address this concern by broadening our research scope to include other languages, such as C# (a statically typed language analogous to Java) and Python (a prevalent dynamically typed language) in future research.

VI. Conclusions

In this work, we propose a set of seven predicates for identifying necessary mock objects. For each predicate, we offer detailed justifications and examples to effectively communicate our intentions. We perform an analysis using five of these predicates on a Java codebase with the help of Doop analysis. We present potential explanations for our empirical results, which can be further studied by the research community. Our future work includes extending our analysis to other programming languages and paradigms to investigate potential variations in mock usage patterns among different practitioner groups.

References

[1] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 1999.

[2] T. Mackinnon, S. Freeman, and P. Craig, "Endo-Testing: Unit Testing with Mock Objects," *Extreme programming examined*, pp. 287–301, 2000.

[3] M. Fowler, "Mocks Aren't Stubs," 2007. [Online]. Available: https://martinfowler.com/articles/mocksArentStubs.html

[4] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[5] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 1999.

[6] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 236–246, 2004.

[7] T. Freese, "EasyMock." [Online]. Available: https://easymock.org/

[8] S. Faber, "Mockito." [Online]. Available: https://site.mockito.org/

[9] R. C. Martin, "When to Mock," 2014. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2014/05/10/WhenToMock.html

[10] J. B. Rainsberger, "Integrated Tests Are A Scam," 2009. [Online]. Available: https://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam

[11] J. Langr, "Don't mock me: Design considerations for mock objects," in *Agile Development Conference*, vol. 2004, 2004.

[12] K. Beck, M. Fowler, and D. H. Hansson, "Is TDD Dead?" 2014. [Online]. Available: https://martinfowler.com/articles/is-tdd-dead/

[13] F. Pereira, "TTDD - Tautological Test Driven Development (Anti Pattern)." [Online]. Available: https://fabiopereira.me/blog/2010/05/27/ttdd-tautological-test-driven-development-anti-pattern/

[14] J. B. Rainsberger, "Who Tests the Contract Tests?" 2018. [Online]. Available: https://blog.thecodewhisperer.com/permalink/who-tests-the-contract-tests

[15] T. Kim, C. Park, and C. Wu, "Mock Object Models for Test Driven Development," *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pp. 221–228, 2006.

[16] D. Saff and M. D. Ernst, "Mock object creation for test factoring," *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 49–51, 2004.

[17] N. Tillmann and W. Schulte, "Mock-object generation with behavior," *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 365–368, 2006.

[18] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "GenUTest: a unit test and mock aspect generation tool," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, p. 273, 2009.

[19] S. J. Galler, A. Maller, and F. Wotawa, "Automatically extracting mock object behavior from Design by Contract™ specification for test data generation," *Proceedings of the 5th Workshop on Automation of Software Test*, pp. 43–50, 2010.

[20] J. Nandigam, V. N. Gudivada, A. Hamou-Lhadj, and Y. Tao, "Interface-Based Object-Oriented Design with Mock Objects," *2009 Sixth International Conference on Information Technology: New Generations*, pp. 713–718, 2009.

[21] K. Taneja, Y. Zhang, and T. Xie, "MODA: automated test generation for database applications via mock objects," *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 289–292, 2010.

[22] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[23] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

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

[25] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 243–262. [Online]. Available: https://doi.org/10.1145/1640089.1640108