ASSESSING MOCK CLASSES: AN EMPIRICAL STUDY

GUSTAVO HENRIQUE ALVES PEREIRA

ASSESSING MOCK CLASSES: AN EMPIRICAL STUDY

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Andre Cavalcante Hora

Belo Horizonte

Setembro de 2021

GUSTAVO HENRIQUE ALVES PEREIRA

ASSESSING MOCK CLASSES: AN EMPIRICAL STUDY

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Andre Cavalcante Hora

Belo Horizonte

September 2021

Pereira, Gustavo Henrique Alves.

P436a Assessing mock classes: an empirical study [manuscrito] / Gustavo Henrique Alves Pereira . – 2021.
xv, 55 f. il.

Orientador: André Cavalcante Hora.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.51-55
.
1. Computação – Teses. 2. Software – Testes –Teses. 3. Software – Validação – Teses. 4. Framework (Programa de computador) – Teses. I. Hora, André Cavalcante. II.Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Assessing Mock Classes: An Empirical Study

GUSTAVO HENRIQUE ALVES PEREIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. ANDRÉ CAVALCANTE HORA - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. BRUNO BARBIERI DE PONTES CAFEO
Faculdade de Computação - UFMS

Belo Horizonte, 2 de Setembro de 2021.

Acknowledgments

Este trabalho foi produto de anos de preparo e dedicação, contando sempre com o apoio de várias pessoas. Agradeço particularmente a:

Minha mãe, Vanessa, que sempre encorajou meu percurso na vida acadêmica.

Meu pai, Nilson, que me incentivou a seguir uma carreira técnica, e tão cedo nos deixou durante o curso deste mestrado.

Minha esposa, Lorena, que apoiou este trabalho desde o princípio, mesmo quando ele ocupou parte do nosso tempo juntos, incluindo nossa celebração de casamento.

Meu mentor e amigo, professor Leandro Maia, inspirador de minha formação em Ciência da Computação, por admirar sua inspiradora didática e ética de trabalho.

Meu orientador, professor André Hora, que acreditou na proposta e sempre ofereceu conselhos valiosos, me guiando com seu senso de excelência e rigor científico.

Aos membros do ASERG, pelas discussões relevantes e ampliadoras de horizontes.

Meus colegas de trabalho, sempre compreensivos, incentivando e se interessando por este trabalho.

Ao DCC/UFMG e CNPq pelo suporte financeiro na participação em eventos, e pela manutenção do PPGCC.

"The programmer should not ask how applicable the techniques of sound programming are, he should create a world in which they are applicable; it is his only *way of delivering a high-quality design."*

(Edsger W. Dijkstra)

Resumo

Em atividades de teste, desenvolvedores frequentemente utilizam dependências que tornam os testes mais difíceis de serem implementados. Neste cenário, eles podem utilizar objetos mock para simular o comportamento de tais dependências, o que contribui para tornar os testes rápidos e isolados. Na prática, as dependências simuladas podem ser dinamicamente criadas com o apoio de frameworks de mock ou manualmente codificadas em classes mock. Enquanto frameworks de mock são bem explorados pela literatura, classes mock ainda são pouco estudadas. Avaliar classes mock pode fornecer as bases para melhor compreender como estes mocks são criados e consumidos por desenvolvedores e para detectar novas práticas e desafios. Nesta dissertação, propomos um estudo empírico e um survey para avaliar classes mock. Ao analisar projetos de software populares na linguagem Java, detectamos mais de 600 classes mock e avaliamos o seu conteúdo, projeto, evolução e utilização. Também conduzimos um survey com 39 desenvolvedores que fizeram manutenção em classes mock para melhor compreender motivações de uso. Descobrimos que classes mock: frequentemente simulam objetos de domínio, dependências externas, e serviços web; são tipicamente parte de uma hierarquia; são em sua maior parte públicas, mas 1/3 são privadas; e são largamente consumidas por projetos clientes, particularmente para apoiar testes de aplicações web. Desenvolvedores argumentam que frameworks de mock podem reduzir a qualidade do código e possuem limitações, enquanto classes mock podem melhorar a qualidade do código e são simples de se configurar em testes. Além disso, desenvolvedores declaram diversas razões sobre quando criarmos classes mock, por exemplo, para testes complexos e para apoiar o reuso; frameworks de mock deveriam, idealmente, serem utilizados para criar testes unitários simples, mas também para testar entidades externas e para testar código com mudanças mínimas. A percepção geral é que a utilização de classes mock é preferível do que os frameworks de mock, entretanto, existem casos específicos onde o uso dos frameworks de mock são uma escolha adequada. Finalmente, fornecemos implicações e observações para pesquisadores e profissionais.

Palavras-chave: Engenharia de Software, Teste de Software, Test Doubles, Frameworks de Mock, Classes Mock.

Abstract

During testing activities, developers frequently rely on dependencies that make the test harder to be implemented. In this scenario, they can use mock objects to emulate the dependencies' behavior, which contributes to make the test fast and isolated. In practice, the emulated dependency can be dynamically created with the support of mocking frameworks or manually hand-coded in mock classes. While the former is well-explored by the research literature, the latter lacks further empirical analysis. Assessing mock classes would provide the basis to better understand how those mocks are created and consumed by developers and to detect novel practices and challenges. In this dissertation, we propose an empirical and a survey study to assess mock classes. We analyze popular Java software projects, detect over 600 mock classes, and assess their content, design, evolution, and usage. We also conduct a survey with 39 developers who maintained mock classes to better understand usage motivations. We find that mock classes: often emulate domain objects, external dependencies, and web services; are typically part of a hierarchy; are mostly public (but 1/3 are private); and are largely consumed by client projects, particularly to support web testing. Developers argue that mocking frameworks may reduce code quality and have limitations, while mock classes may improve code quality and are simple to set up in tests. Moreover, developers state several reasons to create mock classes, for example, for complex testing and to support reuse; mocking frameworks should ideally be used to create simple unit tests, but also to test external entities and to test code with minimal change. The overall perception is that the usage of mock classes is preferable over mocking frameworks, however, there are specific cases in which mocking frameworks are a better choice. Finally, we provide implications and insights to researchers and practitioners working with mock classes.

Palavras-chave: Software Engineering, Software Testing, Test Doubles, Mocking Frameworks, Mock classes.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation and Problem

Software testing is a key practice in modern software development. Often, during testing activities, developers are faced with dependencies (*e.g.,* web services, database, etc.) that make the test harder to be implemented [Meszaros, 2007]. In this scenario, developers can either instantiate these dependencies inside the test or use mock objects to emulate the dependencies' behavior [Spadini et al., 2017, 2019]. The use of mock objects can contribute to make the test fast, isolated, repeatable, and deterministic [Meszaros, 2007]. A test case that, for example, relies on an unstable and slow external service can mock this dependency to be stable and faster. To support the learning of mock objects, technical literature is available for distinct programming languages and ecosystems (*e.g.,* Feathers [2004]; Meszaros [2007]; Freeman and Pryce [2009]; Osherove [2009]; Percival [2014]).

In practice, there are two solutions to adopt mock objects. The emulated dependency can be dynamically created with the support of mocking frameworks or manually hand-coded in *mock classes* [Meszaros, 2007]. Indeed, mocking frameworks are quite popular nowadays and are found in distinct software ecosystems. For example, JavaScript developers may use SinonJS[1] and Jest,[2] which is supported by Facebook; Java developers can rely on Mockito,[3] while Python provides unittest.mock[4] in its core library. The other solution to create mock classes is by hand, that is, manually creating emulated dependencies so they can be used in test cases. In this case, developers do not

[1]`https://sinonjs.org`
[2]`https://jestjs.io`
[3]`https://site.mockito.org`
[4]`https://docs.python.org/3/library/unittest.mock.html`

need to rely on any particular mocking framework, since they can directly consume the mock class. For example, to facilitate web testing, the Spring web framework includes a number of classes dedicated to mocking.[5] Similarly, the Apache Camel integration framework provides mock classes to support distributed and asynchronous testing.[6] In those cases, instead of using a mocking framework to simulate a particular dependency, developers can directly use mock classes on their test cases, such as `MockServer`, `MockHttpConnection`, `MockServlet`, etc.

Past research showed that mocking frameworks are largely adopted by software projects [Mostafa and Wang, 2014] and that they may indeed support the creation of unit tests [Marri et al., 2009; Arcuri et al., 2017; Spadini et al., 2017, 2019]. Moreover, recent research showed how and why practitioners use mocks and the challenges faced by developers [Spadini et al., 2017, 2019]. However, those researches are restricted to the context of mocking frameworks. To the best of our knowledge, mock classes have not yet been studied by the research literature. In this context, some important questions are still open, such as: what dependencies are emulated by those mock classes? are they any different from framework mocks? how are mock classes designed and used by developers? why and when mock classes are adopted instead of mocking frameworks? Thus, assessing mock classes would provide the basis (i) to understand how those mock objects are created and consumed by developers and (ii) to detect novel practices and challenges.

1.2 Proposed Work

In this dissertation, we propose an empirical and a survey study to assess mock classes. *First*, we analyze 12 popular Java software projects hosted on GitHub (including Elasticsearch, Spring Boot, and Google Guava) and detect 604 mock classes. We then propose the following research questions to assess the content, design, evolution, and usage of those mock classes:

- *RQ1 (Content): What is the content of mock classes?* We manually categorize the 604 mock classes with respect to the dependency they are emulating. We observe that mock classes typically emulate domain objects, external dependencies, and web services. The most and least frequent categories are essentially the same that developers create when relying on mocking frameworks.

[5]https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html#mock-objects

[6]https://camel.apache.org/components/latest/mock-component.html

- *RQ2 (Design): How are mock classes designed?* We assess the structural details of the mock classes. We find that mock classes often extend other classes or implement interfaces; around 70% are public and can be reused, while 30% have restrictive visibility; and mock and regular classes have the same number of methods.

- *RQ3 (Evolution): How do mock classes evolve over time?* Overall, mock classes and regular classes have equivalent number of changes. However, there may exist exceptions.

- *RQ4 (Usage): How are mock classes used by developers?* In this analysis, we focus on the client-side. With the support of the Boa platform [Dyer et al., 2013], we assess millions of client projects and detect that mock classes are largely consumed, particularly to emulate web services. However, the usage is very concentrated on a few classes: 10 classes are consumed by 76% of the clients.

Second, to understand the motivations behind the usage of mock classes, we conduct a survey with 39 developers who maintained this kind of class. We focus on assessing *why* and *when* developers rely on mock classes. Thus, we propose two research questions:

- *RQ5 (Motivation): Why do developers rely on mock classes instead of mocking frameworks?* We find that that developers rely on mock classes over mocking frameworks due to several reasons. They argue that mocking frameworks reduce code quality, have limitations, and add dependencies. Also, developers mention that mock classes improve code quality and are simple to set up in tests.

- *RQ6 (Choice): When should the developer create mock classes and when is it better to rely on mocking frameworks?* According to the surveyed developers, there are many reasons to create mock classes: for complex testing, to avoid mocking framework limitations, to have better code quality, to support reuse, and to keep project quality. On the other hand, developers should rely on mocking frameworks mainly to create simple unit tests, but also to have better code quality and to test external entities.

Based on our results, we provide insights and practical implications to researchers and practitioners by discussing topics as (i) the novel empirical data on mock classes, (ii) the reusability and lack of visibility of the mock classes, (iii) the widespread usage of the mock classes, and (iv) the reasons for using mock classes and mocking frameworks.

We reveal novel quantitative and qualitative empirical data about the creation of mock classes, which can guide practitioners in charge of maintaining them. We shed light on the over creation of private mock classes, which can be harmful to the overall project maintainability. We present that the usage of mock classes is not restricted to the target projects of this study, but it seems to be widespread, thus, they should be maintained with care because client projects can be impacted. Lastly, we find that the overall developers' perception is that the usage of mock classes is preferable over mocking frameworks, however, there are specific cases in which mocking frameworks are perceived as a better choice.

1.3 Contributions and Publications

The contributions of this research are summarized as follows:

- We provide the first empirical study on mock classes, from both quantitative and qualitative perspectives.
- We perform an empirical analysis of mock classes to better understand their content, design, evolution, and usage.
- We perform a survey analysis with practitioners to understand their decision process when deciding why and when they use mock classes and mocking frameworks.
- We propose insights and practical implications to researchers and practitioners working on and consuming mock classes.

The research reported in this dissertation has generated the following publication:

- Gustavo Pereira, Andre Hora. Assessing Mock Classes: An Empirical Study. IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 453-463, 2020 (`https://doi.org/10.1109/ICSME46990.2020.00050`)

1.4 Dissertation Outline

The remaining of this dissertation is organized as follows:

- Chapter 2 presents the background for this research, describing mocks, test doubles, mock classes, and related work.

- Chapter 3 details the study design, presenting the adopted projects and how we selected the mock classes.
- Chapter 4 describes the empirical results performed on mock classes.
- Chapter 5 presents the results of the conducted survey, to better understand the reasons mock classes are adopted.
- Chapter 6 presents the results and conclusions of this work.

Chapter 2

Background and Related Work

In this chapter, we provide the background for the study in this dissertation. First, we present an outline on software testing on Section 2.1. Then, we introduce test doubles on Section 2.2. Next, we discuss mocking frameworks (Section 2.3) and mock classes (Section 2.4). We also discuss the related work in Section 2.5. Finally, we conclude the chapter in Section 2.6.

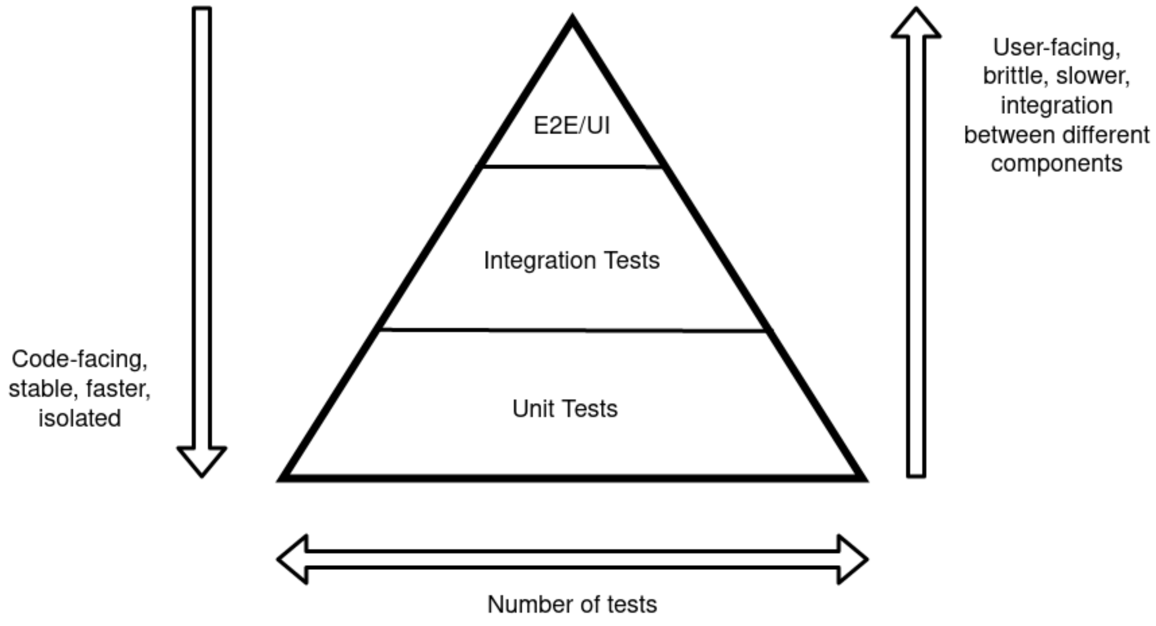
2.1 Software Testing

Software testing is the practice of ensuring that a software is behaving the way it was intended to [Bertolino, 2007]. Myers et al. [2011] defines it as the process of executing a program with the intention of finding errors. Within the context of software being expected to be shipped constantly, agile development practices such as continuous deployment arose, and so the necessity for software to work as expected grew [Freeman and Pryce, 2009].

The literature considers a variety of test definitions, according to their goals and techniques. One convenient definition for automated testing is the Test Pyramid [Cohn, 2009]. This model, as presented on Figure 2.1, represents automated tests in a layered fashion. Unit tests are at the base, meaning they should exist in a larger number. End-to-End tests (E2E), known as well as User Interface (UI) tests, are at the top, representing that they should exist in lower quantities. While introduced by Cohn [2009], it was further improved and discussed by other authors [Vocke, 2018; Valente, 2020; Winters et al., 2020; Khorikov, 2020]. Each layer represents a level of test automation, and are defined as follows:

- *Unit Tests* are used to automatically verify small code units. Due to being created

Figure 2.1: The Test Pyramid.


E2E/UI
Integration Tests
Unit Tests
Code-facing, stable, faster, isolated
User-facing, brittle, slower, integration between different components
Number of tests

to examine such small units, they should run quickly and be stable. Khorikov [2020] defines them as automated tests that verify a small piece of code, do it quickly, and in an isolated manner. Because of this, they are on the base of the pyramid, being the most numerous tests.

- *Integration Tests* aim to check a functionality or transaction of a system in a combined way. They verify whether the code works integrated with external dependencies, being those other parts of the same system, other processes, or even in other machines, via networks. Since they need to invoke or simulate such dependencies, integration tests are more vulnerable to external change, and may be slower to run. For this characteristic, they are above the unit tests' layer on the pyramid.

- *E2E/UI* simulate a system's usage as closely as possible to the real, user-facing way. As they are used to validate the whole system, E2E tests may be even considered as a special case of integration tests, in a broader scope. As the UI is more prone to be changed comparing to underlying code or APIs, they tend to be more fragile. Being tests that validate possibly complex visual behavior and need to wait for all the system's levels to be ran in order to be tested, they are also the slowest ones. Because of this, E2E tests are in the category with the lowest number of tests.

The higher a level is located on the pyramid, the more brittle, slower and expensive are the tests contained in it. The individual units referred in *unit testing* may be classes, methods, or any compatible definition [Myers et al., 2011]. Such tests are often seen as a way to keep software organized during the lifetime of a project, preventing regressions and architectural degradation [Khorikov, 2020], and even as a precondition for refactoring [Fowler, 2018]. This practice is widely supported in modern programming languages, such as Java,[1] C#,[2] Python,[3] and Go,[4] to cite a few examples.

2.2 Test Doubles

Test double is the broadest term to describe any fake thing introduced in place of a real thing for the purpose of writing an automated test.[5] Test double replaces a component on which the System Under Test (SUT) depends with a test-specific equivalent [Meszaros, 2007]. Formally, a mock object is a particular type of test double [Meszaros, 2007]. In addition to mock objects, other test doubles are dummies, stubs, spies, and fake objects, each one with its nuances [Meszaros, 2007].

Despite the formal definitions, the state of the practice is to frequently use the terms mock objects and test doubles interchangeably, for example:

- Robert Martin (author of Clean Code [Martin, 2009]): "*The word "mock" is sometimes used in an informal way to refer to the whole family of objects that are used in tests.*"[6]

- Martin Fowler (author of Refactoring [Fowler, 2018]): "*The term Mock Objects has become a popular one to describe special case objects that mimic real objects for testing.*"[7]

- Harry Percival (author of TDD with Python [Percival, 2014]): "*I'm using the generic term "mock", but testing enthusiasts like to distinguish other types of a general class of test tools called Test Doubles [...] The differences don't really matter for this book.*"[8]

[1]https://junit.org/junit5/
[2]https://xunit.net/
[3]https://docs.pytest.org/en/6.2.x/
[4]https://golang.org/pkg/testing/
[5]https://github.com/testdouble/contributing-tests/wiki/Test-Double
[6]https://blog.cleancoder.com/uncle-bob/2014/05/14/TheLittleMocker.html
[7]https://martinfowler.com/articles/mocksArentStubs.html
[8]http://www.obeythetestinggoat.com/book/chapter_mocking.html

- Vladimir Khorikov (in Unit Testing: Principles, Practices and Patterns [Khorikov, 2020]): "*[The] main thing to remember is that mocks are a subset of test doubles. People often use the terms test double and mock as synonyms, but technically, they are not.*"
- testdouble.js wiki about testing: "*There are several sub-types of test doubles, but most tools do a poor job either following a conventional nomenclature [...].*"[9]

The terminology around the kinds of test doubles is confusing and inconsistent, hence different people use distinct terms to mean the same thing. As a result, this leads to an endless discussion[10] on such theme. Nevertheless, since literature on the theme [Meszaros, 2007; Fowler, 2007] presents the aforementioned distinction on the kinds of test doubles, it is fundamental to better describe it here:

- *Dummy.* Objects that are not properly used, rather just referenced. They are commonly employed to fill in parameters for methods that may require them. They may be as simple as null objects or instances of the *Object* class, for example [Meszaros, 2007].
- *Stub.* Objects that provide pre-programmed answers for method calls. Usually they are created for some specific tests, not being general-purpose testing objects [Meszaros, 2007].
- *Spy.* Considered a type of *Stub*, Spy objects are similar to those but with the capability to monitor values in their interactions with the SUT, which may help on debugging or some finer kind of assertion [Meszaros, 2007].
- *Mock.* The most known term of those, such objects are similar to *Stubs* in the way that they are programmed to return some specific values, but they are characterized by the emphasis on verifying indirect outputs, or behavior, of the SUT [Meszaros, 2007].
- *Fake.* Objects that actually have working implementations but are far simpler than the real dependency of the SUT. As they are specifically built for testing, such objects are not production-ready [Meszaros, 2007].

Since the usage of the "mock" term for test double is already well established in the literature, we will keep using it in this work.

[9] `https://git.io/JRWeJ`
[10] Example: `https://bit.ly/3d1XumC`

2.3 Mocking Frameworks

Framework is a designation for a partially complete software subsystem intended for instantiation, defining an architecture for a family of subsystems by providing the basic building blocks to create them, defining points on which adaptations can be attached upon [Buschmann et al., 1996]. The definition from Johnson [1992] is that a framework is a reusable design for solutions to problems in some particular problem domain. Since reusability is a key point on software construction, frameworks perform a role on both avoiding code repetition and embedding standard practices.

A mocking framework provides functionality for making the creation of test doubles simpler on a unit testing context. It allows the specification of interactions between objects, specifying test entities and a variety of other facilities. For Java, Mockito [Mockito, 2021] is a well-known open-source framework that promises to create clean and readable tests with a simple API and clear verification errors. To facilitate the creation of mock objects, Mockito provides the following basic APIs:

- `mock()/@Mock` is used to create mocks, and the `when()` and `given()` methods for specifying how a mock should behave.
- `spy()/@Spy` for partial mocking, with real methods for the mocked classes being invoked but still being able to be verified and stubbed.
- `@InjectMocks` to automatically inject mocks or spies to placess annotated with `@Spy` or `@Mock`.
- `verify()` to check whether methods were called with some given arguments.

In the Java community, other popular mocking frameworks are jMock [jMock, 2007] and EasyMock [EasyMock, 2021]. jMock itself dates back to the emerging of the practice, being introduced by Freeman et al. [2004]. As for EasyMock, its authors are nominally cited in Mockito's page "*for their ideas on beautiful and refactoring-friendly mocking API. First hacks on Mockito were done on top of the EasyMock code*".

In Python, the standard test library `unittest` contains a `mock` module [Python Software Foundation, 2020]. It is described as allowing developers to "*replace parts of your system under test with mock objects and make assertions about how they have been used*". As it is present on the language's core libraries, it makes easier for users to be pointed to its usage, without the need for looking to third-party packages.

In the JavaScript ecosystem, alternatives such as Jest [Facebook, 2021] and Sinon.JS [Sinon.js, 2021] can be adopted to create mock objects. Jest is a testing framework that contains a mocking component, called Mock Functions, described as allowing to "*test the links between code by erasing the actual implementation of a function, capturing calls to the function (...) and allowing test-time configuration of return values*". They also allow mocking functions to be used in test code or support writing manual mocks to override module dependencies. Sinon.JS is a tool for standalone test spies, stubs, and mocks, claimed to work with any unit testing framework for JavaScript. Its API provides methods for mocking objects and verifying behavior. Since JavaScript programming is heavily skewed to the usage of functions, it is interesting to notice the focus on mocking functions in the languages' frameworks.

In C#, Moq [moq, 2021] claims to be the most popular mocking library for .NET. Its concept is similar to those of other languages, with a motivation to be used by developers not previously using mocking libraries and were manually writing their own mocks. Noticing the high barrier of entry of other mocking libraries, the authors decided to build a lightweight approach, while taking advantage of some advanced features of the .NET platform, such as LINQ.[11]

Figure 2.2 presents examples of mock objects created by Mockito in Java projects Elasticsearch and Spring Boot. First, Figure 2.2a shows the method `setupMocks()`, inside the class `TransportFinalizeJobExecutionActionTests`.[12] This is a method that prepares mocks to be used on other tests. The `client` attribute is being instantiated with a mock of the `Client` class, via Mockito's `mock()` method. Then, Mockito's `doAnswer()` method is called to stub an answer for when a method is called, with the `when()` method being used to prepare the particular value it needs to answer. Other uses of stubbing are being done in the method via `when()`. Figure 2.2b displays another example on the same class, with method `createAction()` instantiating a `TransportFinalizeJobExecutionAction` with mocked instances of some of its parameters. Figure 2.2c represents method `accessManagerVetoRequest()` in the class `DispatcherTests` from project Spring Boot.[13] As the name of the class indicates, it is used for tests of the Dispatcher entity. The method in question is a test of a forbidden HTTP response (403), which happens after a configuration for the system to provide such a response. Objects of the `HandlerMapper` and `Handler` classes are being created via Mockito's `mock()` API for further usage.

[11] `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/`
[12] https://git.io/JRCQL
[13] https://git.io/JRC5l

(a) setupMocks - Elasticsearch

```java
private void setupMocks() {
    ExecutorService executorService = mock(ExecutorService.class);
    threadPool = mock(ThreadPool.class);
    org.elasticsearch.mock.orig.Mockito.doAnswer(invocation -> {
        ((Runnable) invocation.getArguments()[0]).run();
        return null;
    }).when(executorService).execute(any(Runnable.class));
    when(threadPool.executor(MachineLearning.UTILITY_THREAD_POOL_NAME)).thenReturn(executorService);

    client = mock(Client.class);
    doAnswer( invocationOnMock -> {
        ActionListener listener = (ActionListener) invocationOnMock.getArguments()[2];
        listener.onResponse(null);
        return null;
    }).when(client).execute(eq(UpdateAction.INSTANCE), any(), any());

    when(client.threadPool()).thenReturn(threadPool);
    when(threadPool.getThreadContext()).thenReturn(new ThreadContext(Settings.EMPTY));
}
```

(b) createAction - Elasticsearch

```java
private TransportFinalizeJobExecutionAction createAction(ClusterService clusterService) {
    return new TransportFinalizeJobExecutionAction(mock(TransportService.class), clusterService,
            threadPool, mock(ActionFilters.class), mock(IndexNameExpressionResolver.class), client);

}
```

(c) accessManagerVetoRequest - Spring Boot

```java
@Test
void accessManagerVetoRequest() throws Exception {
    given(this.accessManager.isAllowed(any(ServerHttpRequest.class))).willReturn(false);
    HandlerMapper mapper = mock(HandlerMapper.class);
    Handler handler = mock(Handler.class);
    given(mapper.getHandler(any(ServerHttpRequest.class))).willReturn(handler);
    Dispatcher dispatcher = new Dispatcher(this.accessManager, Collections.singleton(mapper));
    dispatcher.handle(this.serverRequest, this.serverResponse);
    verifyNoInteractions(handler);
    assertThat(this.response.getStatus()).isEqualTo(403);
}
```

Figure 2.2: Examples of mocking frameworks usage.

2.4 Mock Classes

As briefly described in the introduction, mock objects can be either (1) dynamically created with the support of mocking frameworks or (2) manually hand-coded in mock classes [Meszaros, 2007]. For example, suppose a developer wants to simulate a dependency on `HttpRequest` when performing web testing. In this case, the developer could either use a mocking framework to dynamically create a mock object that simulates

`HttpRequest`, or create a mock class by hand (*e.g.,* `MockHttpRequest`) to simulate `HttpRequest`. Interestingly, writing the mock classes forces developers to give those mocks names, so they can be reused in other tests; moreover, mock classes should be somehow designed to be consumed by clients [Martin, 2014].

To avoid the inconsistency around mock objects and test doubles previously stated, we rely on a lightweight solution to detect mock classes. We consider mock classes the ones that are explicitly declared as mocks, that is, the classes with the term "mock" in their names. For example, the class `MockServerHttpRequest`[14] provided by the Spring framework is a mock implementation of the real `ServerHttpRequest`. Thus, due to the lack of consistency in the terminology, we recognize that we may detect other test doubles and not only mock objects. Consequently, in this study, the usage of the term mock resembles the widespread usage in the larger field of software development, that is, mocking being adopted as a generic term to represent any test double.

Figure 2.3 presents examples of real mock classes in projects Apache Camel and Apache Dubbo. First, Figure 2.3a shows the mock class `MockMaskingFormatter`.[15] It defines the attribute `received` and implements the method `format()` of the interface `MaskingFormatter`, which sets the attribute. This mock class supports the creation of tests in two test classes (`LogMaskTest` and `LogEipMaskTest`). Next, Figure 2.3b presents the mock class `MockedExchangeHandler`,[16] which implements two methods that only throw an exception. Notice that this mock class is private, thus, it is only used internally in the same test file it was defined and cannot be reused by other tests. Lastly, the mock class `CloudWatchClientMock`[17] in Figure 2.3c implements three methods of the interface `CloudWatchClient`. This mock class is then used to support the creation of several tests in project Apache Camel, for example, in class `CwComponentRegistryClientTest`.[18]

Interestingly, mock classes have their own characteristics. For instance, they may have distinct goals, visibility, number of methods, and usage of inheritance or interfaces, to name a few. The study of these differences may lead to a deeper understanding on how a substantial part of automated tests are written.

[14]`https://bit.ly/3bQF9ZH`
[15]`https://git.io/JlTff`
[16]`https://git.io/JlTfL`
[17]`https://git.io/JlTfq`
[18]`https://git.io/JlTJJ`

2.5 Related Work

Spadini et al. [2017] state that mock objects are common when testing software dependencies and their use is supported by a variety of frameworks in several programming languages. The authors present that, despite this practice being common, there is a lack of empirical knowledge on how and why practitioners use mocks. To this aim, the study detects classes that are mocked with the Mockito framework in Java projects and categorize them. Moreover, the authors interview developers to understand the reasons for mocking dependencies and the main challenges they face. The study finds that the dependencies that are most mocked are those that make testing difficult and that classes that are in complete control of the developers are least mocked; the challenges are related to technical issues such as coupling between the mock and the production code. The authors then extended their study with investigations about mocks introduction and evolution and expanded code quality metrics evaluation [Spadini et al., 2019]. The new research confirms the findings of their previous study [Spadini et al., 2017] in the sense developers tend to mock dependencies that make tests more difficult, and show that mocks usually evolve together with the test classes, being added at the beginning of their history and changing accordingly. In our study, we focus on *mock classes* instead of mock objects that are mocked via mocking frameworks like Mockito. Therefore, the results presented in this paper complement previous ones with respect to mocking frameworks.

Meszaros [2007] provides an introduction and discussion about test doubles, presenting a taxonomy based on their purpose and usage. This taxonomy is further explored and discussed by Fowler [2006, 2007]. The creation of manual mock classes versus the automatic creation via frameworks is also discussed by Robert Martin, stating that developers write their own mocks to improve reuse, project design, and performance due to not using reflection [Martin, 2014]. Interestingly, we also find reuse among the reasons to adopt mock classes in RQ6. In contrast, some authors observe a disadvantage in using mocks in general. Elliot discusses how mocks can be code smells by relating them to tight coupling, a symptom of code smells itself, particularly in the context of functional programming [Elliott, 2017]. The author states that the drive for mocking in unit tests is to achieve complete test coverage but there is a flawed decomposition strategy, which causes the need for mocks. Winters et al. [2020] state that real implementations should be preferred while testing, and specifically over some kinds of test doubles citing that their overuse make tests unclear and brittle.

There is a vast technical literature about test doubles [Meszaros, 2007; Feathers, 2004; Freeman and Pryce, 2009; Osherove, 2009; Percival, 2014]. Earlier uses of the term “mocking” can be traced back to the XP community. For example, Mackinnon et al. [2000] present that mock help writing tests in isolation for non-trivial code. Freeman et al. [2004] improve this notion by taking the focus from isolating tests to guiding the discovery of a coherent system of types within a code base, and also introducing the mocking library jMock.[19] The notion of having to test dependencies that are not yet available is also explored by Thomas and Hunt [2002].

Bhagya et al. [2019] generate HTTP services mocks that can provide service responses suitable for testing. Sivashanmugam and Palanisamy [2009] present an approach on testing SQL server integration with mock objects. Mostafa and Wang [2014] study mocking frameworks in software testing, showing a wide usage in this discipline and calling for more studies on the topic. Arcuri et al. [2017] assess the role of mock objects in automatic unit test generation, stating that its use in the EvoSuite[20] tool helps increasing branch coverage and fault detection.

The literature emphasizes that testing code (as any other code) should be maintained as software evolve [Martin, 2009]. This way, issues related to code maintenance as code smells [Fowler, 2018] are valid in tests as well, as test smells [Tufano et al., 2016; Palomba and Zaidman, 2017; Spadini et al., 2018b; Bavota et al., 2015]. Therefore, if one considers that mock classes are part of the tests themselves, maintaining test suite implies maintaining the mock classes as well. In this context, there is a vast literature on testing maintenance. For example, van Bladel and Demeyer [2020] study the issue of duplicated test code by comparing the results of four different clone detection tools, observing between 23% and 29% test code duplication found by the them, but noticing that most tools suffer from false negatives. Zaidman et al. [2011] examine the co-evolution of production and test code, noticing periods of intense testing in development history and the fraction of test code increasing alongside test coverage.

Although few research studies assess mock objects, they focus on the perspective of mocking frameworks. Our study assesses the mock classes created by hand to support testing, thus we complement the existing literature on mock objects. Moreover, we provide several insights on the reasons developers rely on mock classes instead of mocking frameworks.

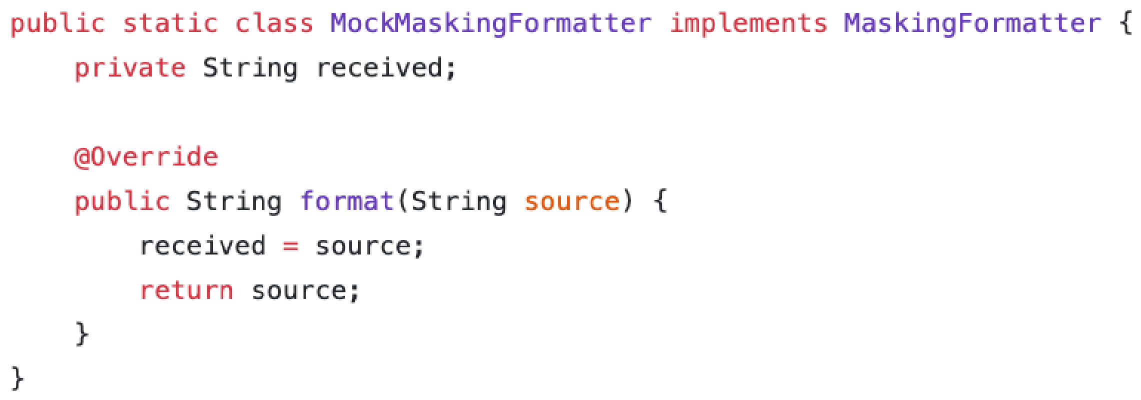
2.6 Final Remarks

In this chapter, we presented base concepts that are used in this dissertation. We started by presenting an overview of software testing, then proceeded by introducing

[19]`http://jmock.org`
[20]`www.evosuite.org`

test doubles. We showed the concept of mocking frameworks, and then displayed the main theme of mock classes: test doubles that are manually implemented as classes in object-oriented projects. We also explored the related literature on the theme. In the next chapters, we will examine aspects of maintainability, evolution and usage of mock classes with our empirical and survey studies.

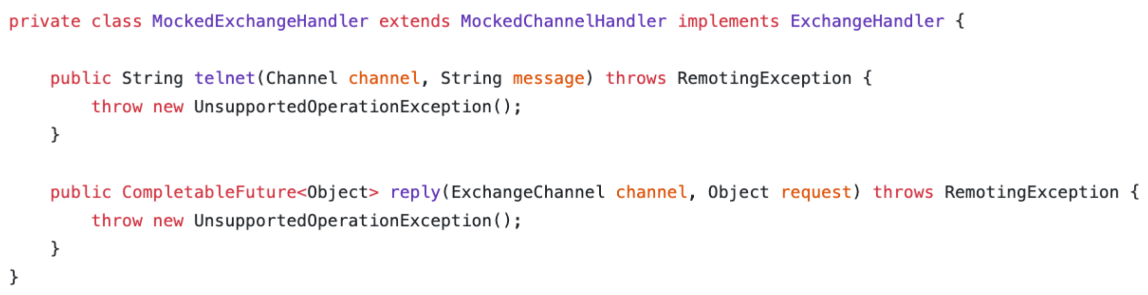
(a) MockMaskingFormatter - Apache Camel



```java
public static class MockMaskingFormatter implements MaskingFormatter {
    private String received;

    @Override
    public String format(String source) {
        received = source;
        return source;
    }
}
```

(b) MockedExchangeHandler - Apache Dubbo

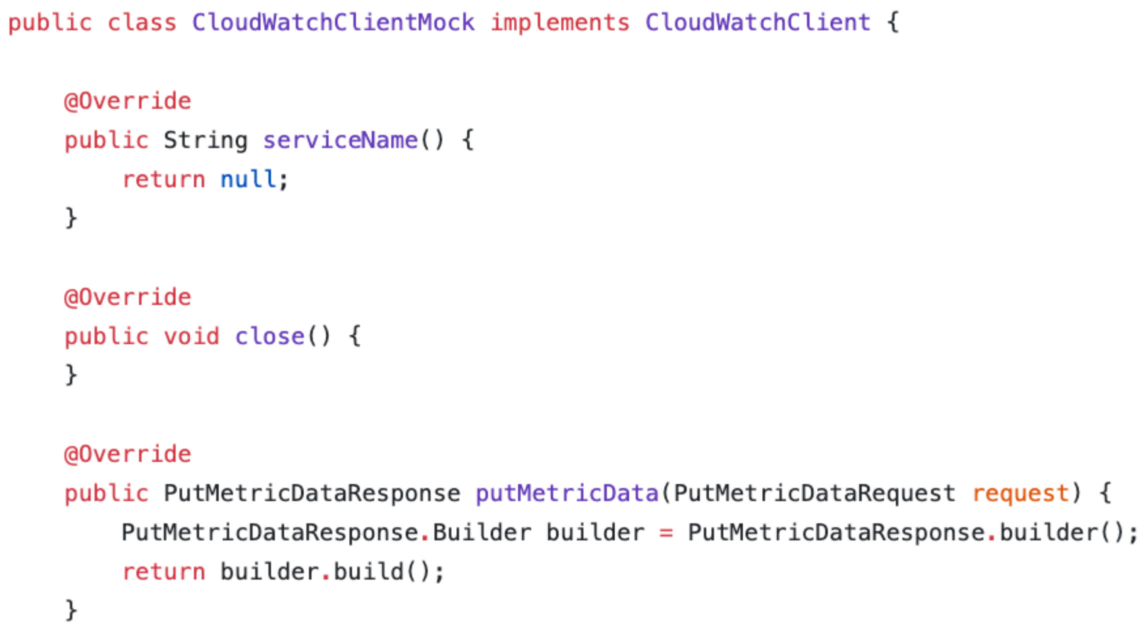


```java
private class MockedExchangeHandler extends MockedChannelHandler implements ExchangeHandler {

    public String telnet(Channel channel, String message) throws RemotingException {
        throw new UnsupportedOperationException();
    }

    public CompletableFuture<Object> reply(ExchangeChannel channel, Object request) throws RemotingException {
        throw new UnsupportedOperationException();
    }
}
```

(c) CloudWatchClientMock - Apache Camel



```java
public class CloudWatchClientMock implements CloudWatchClient {

    @Override
    public String serviceName() {
        return null;
    }

    @Override
    public void close() {
    }

    @Override
    public PutMetricDataResponse putMetricData(PutMetricDataRequest request) {
        PutMetricDataResponse.Builder builder = PutMetricDataResponse.builder();
        return builder.build();
    }
```

Figure 2.3: Examples of mock classes.

Chapter 3

Study Design

In this chapter, we present the study design for the dissertation. Section 3.1 presents how the analyzed projects were found and chosen. Section 3.2 presents how the mock classes were detected in the source code of the projects. In Section 3.3 we present the applied methodology. Section 3.4 describes the research questions. Finally, Section 3.5 contains an ending discussion to the chapter.

3.1 Selecting the Software Projects

In this study, we aim to assess mock classes provided by real world and relevant software projects. We then select the 10 most popular Java systems hosted on GitHub based on the star metric [Borges et al., 2016; Silva and Valente, 2018], which is a proxy for popularity. In addition, two other important projects were included: Apache Lucene-Solr and Apache Camel. The 12 selected projects are presented in Table 3.1. The most popular project of the sample is Elasticsearch (48.3k stars), while the least popular one in it is Camel (3.2k stars). The selected projects cover distinct software domains: web framework (Spring Boot and Spring Framework), search engine (Elasticsearch and Lucene-Solr), asynchronous/event library (RxJava and EventBus), HTTP client (OkHttp and Retrofit), Java support library (Guava), RPC framework (Dubbo), integration framework (Camel), and analytics engine (Spark).

Our dataset is publicly available online at: `https://git.io/Jle3l`.

Table 3.1: Selected projects and detected mock classes.

Project	Stars	Classes	Classes Mocked Using Mockito	Mock Classes
Elasticsearch	48.3k	16,097	524	138
Spring Boot	47.1k	7,845	367	33
RxJava	42.4k	2,766	21	0
Guava	36.8k	6,174	19	12
OkHttp	36.8k	210	0	1
Spring Fw.	36.5k	10,980	272	127
Retrofit	35.5k	303	0	4
Dubbo	32k	2,104	60	58
Spark	25.8k	1,850	23	0
EventBus	22.4k	163	0	0
Lucene-Solr	3.4k	11,583	35	132
Camel	3.2k	18,757	198	99
Total	-	78,832	1,519	604

3.2 Detecting Mock Classes

The next step is to detect mock classes. First, for each project, we extract all classes in the Java files, including nested ones. This result is presented in the column "Classes" in Table 3.1. The largest project in number of classes is Camel (18,757 classes), while the smallest is EventBus (163 classes); in total, those projects include over 78.8k classes. We then perform two analyses to assess (1) mocks created with the support of frameworks and (2) mock classes.

1. Mocks created with the support of mocking frameworks. Before presenting the mock classes, for comparison purposes, we first present the mocks created with the support of mocking frameworks. We verify the number of classes that are mocked using the most popular mocking framework in Java, Mockito (column "Classes Mocked Using Mockito" in Table 3.1).[1] To this aim, we verify the classes being mocked with the Mockito API `Mockito.mock()`, as done by the related literature [Spadini et al., 2017, 2019]. For instance, Elasticsearch mocks 524 classes via Mockito, while Spring Boot mocks 367; in total, we find 1,519 classes being mocked with this framework. Only three systems (OkHttp, Retrofit, and EventBus) do not create any mock with Mockito. This shows that the usage of mocking is widespread to support testing.

2. Mock classes. Lastly, in the column "Mock Classes" in Table 3.1, we present the

[1]We only verified Mockito because it is the *de facto* mocking framework in Java. For comparison, Mockito has over 10,000 stars on GitHub, while EasyMock has 650 stars and JMock has only 119.

target classes of this study, that is, the mock classes. We consider that a class is a mock when:

- It is explicitly declared as a mock class, that is, its name includes the term "mock", but not "mockito" *and*
- It does not use the most popular mocking framework in Java (*i.e.,* it does not import any Mockito class) *and*
- It is not a test class, that is, its name does not end with the suffix "Test" nor "Tests" (which is the guideline to create testing classes in Java [Spadini et al., 2017, 2019]).

As discussed in Section 2.2, the most common term used for referring to test doubles is "mock". Thus, despite the possibility of other kinds of test doubles being created, we only examine classes that contain this term. Therefore, we believe to cover most usage without much loss of detail. Following this approach, we detect 604 mock classes. Six projects have over 30 mock classes: Elasticsearch (138), Camel (99), Spring Framework (127), Lucene-Solr (132), Dubbo (58), and Spring Boot (33). Three projects have 1, 4, and 12 mock classes, while three projects have none.

Table 3.2 presents some examples of mock classes. As we can observe, the class names are very distinct and the dependencies they are simulating are diverse. For example, based on their names, they are likely to be related to web services (*e.g.,* `MockWebSession` and `MockMvc`), network services (*e.g.,* `MockSocketChannel` and `MockMockTelnetHandler`), external dependencies (*e.g.,* `MockGitHub` and `AmazonECS-ClientMock`), to name a few.

Table 3.2: Examples of mock classes.

Project	**Examples**
Elasticsearch	MockSocketChannel, MockBlobStore, MockMessage
Spring Boot	MockCachingProvider, MockFilter, MockServlet
Guava	MockCallback, MockRunnable, MockExecutor
Spring Fw.	MockConnection, MockWebSession, MockMvc
Retrofit	MockGitHub, MockRetrofit, MockRetrofitIOException
Dubbo	MockTelnetHandler, MockDirectory, MockThreadPool
Lucene-Solr	MockTrigger, MockScorable, MockTimerSupplier
Camel	AmazonECSClientMock, MockRest, MockEndpoint

Table 3.3 presents the most frequent terms present on the names of the 604 mock classes (after removing the term “Mock” from them), in order to display which kinds of concerns such mock classes address.

Table 3.3: Mock-related terms.

Pos	**Term**	#	**Pos**	**Term**	#
1	Client	55	6	Http	31
2	Filter	36	7	Script	27
3	Amazon	33	8	Mvc	27
4	Factory	33	9	Server	25
5	Request	33	10	Service	25

We notice the presence of some broader, generic terms such as Client, Filter, and Service, that can refer to a variety of projects and contexts. These show that mock classes are used in multiple roles. We also see some specific ones as Amazon, Http, and Mvc, that are more specific relating to companies, protocols and architectural styles, for example. The top present terms are Client (55 occurrences), Filter (36), Amazon (33), Factory (33), and Request (33), that combine broader and more specific concerns.

3.3 Survey Study

To understand the motivations behind the usage of mock classes and mocking frameworks, we conduct a survey with developers of open-source projects. Specifically, we aim to assess *why* and *when* developers rely on mock classes instead of mocking frameworks. To find candidates to answer to our survey, we mine developers who modified Java files with mock classes and collect their email addresses publicly available on GitHub. We propose a survey with three questions for the developers, as follows:

Dear {author name},
I am a researcher working on software testing and mock objects. In my research, I am studying the mock classes of project {project name}.
I found that you changed the following mock class on file {file name}:

- Class name: {class name}
- Commit date: {commit date}
- Commit link: {commit link}

Could you please answer the following questions:

1. What is the goal of this mock class?
2. Why has a mocking framework (*e.g.,* Mockito) not been used instead?
3. Based on your experience, when should developers create mock classes and when should they rely on mocking frameworks?

Please, provide any other comments or insights regarding the usage of mock classes vs. mocking frameworks.

Question 1 asks for insights about the goal of the mock class. The aim of Question 2 is to put some light on the problem of deciding why to use a mock class in the place of a mocking framework. Lastly, Question 3 seeks to understand the developers' point of view, based on their prior experience.

Initially, we select candidate developers from the studied 12 projects. Given the low amount of candidate developers, we decide to include other popular open-source software projects, based on the GitHub star metric [Borges et al., 2016; Silva and Valente, 2018]. For this purpose, we select software projects until we collect 1,000 unique emails. This way, we add 63 novel projects to our dataset in addition to the original 12 ones. After removing invalid emails and others that refer to the same developer, 658 valid addresses remain. Finally, we are able to successively send 581 emails with our survey and receive 39 responses (7% of response rate).

Finally, as a solution to analyze the developers' responses quantitatively, we rely on *thematic analysis* [Cruzes and Dyba, 2011]. In this analysis, we aim to identify and record themes in textual documents, using the following steps: (1) initial reading of the responses, (2) generating a first code for each response, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. The first three steps are performed by the author, while steps 4 and 5 are done by the author and the advisor until consensus is achieved.

3.4 Research Questions

3.4.1 RQ1 (Content)

Our first research question is intended to assess the content of the mock classes, that is, what type of dependency they are simulating. We rely on the categories proposed by a previous related study in the context of mocking frameworks [Spadini et al., 2017, 2019]. This has two advantages: (1) we keep consistency with previous research studies and (2) we can directly compare our results with previous ones. Thus, we adopt the categories proposed by Spadini et al. [2017, 2019]: domain object, database, native Java libraries, web service, external dependency, and test support. Besides, after inspecting our dataset, a new category is considered: network service. Those categories are defined as follow:

- *Domain object:* classes that are mock to business rules of the system.
- *Database:* classes that are mock to SQL/NoSQL database libraries.
- *Native Java libraries:* classes that are mock to the Java native libraries.
- *Web service:* classes that are mock to some web action.
- *External dependency:* classes that are mock to external libraries.
- *Test support:* classes that support testing itself.
- *Network service:* classes that support network services.

To categorize the mock classes, we perform a qualitative analysis based on (1) class names, (2) documentation analysis, and (3) source code inspection. Some mock classes can be resolved based on the analysis of their names. For example, `MockWebServer`, `MockMvc`, and `MockRest` are examples of the category web service, while `MockSocketChannel`, `MockTcpChannelFactory`, `MockTcpReadWriteHandler` are examples of network service. Other classes, such as `MockTokenizer`, `MockTargetSource`, and `MockAnalyzer` are not straightforward and require either documentation analysis or code inspection to detect their categories.

3.4.2 RQ2 (Design)

In this second research question, we aim to study how mock classes are designed. We then assess some important structural aspects, as follows:

- *Inheritance and Interface*: mock classes can be created by extending classes or implementing interfaces. We assess whether the mock classes are more likely to implement interfaces, extend classes, or none of them. Our final goal is to better understand whether mock classes are part of hierarchies or are standalone classes.
- *Visibility*: mock classes can have a public scope to be used all over the project (or by external clients) or they can have private scope to be only locally used. Thus, we explore the visibility of the mock classes to verify whether they are likely to be reusable or not.
- *Number of methods*: we analyze the number of methods provided by the mock classes. We also compare with the number of methods provided by regular classes, which are randomly selected following the same distribution of mock classes per system. We aim to understand whether mock classes are likely to be larger (which may demand more effort to create and maintain) or smaller (which may demand less effort).

3.4.3 RQ3 (Evolution)

In the third research question, we focus on understanding how mock classes change over time. As mock classes emulate dependencies, they are likely to change whenever the dependencies change [Spadini et al., 2017]. This happens because the mock class should remain compatible with the original behavior. However, so far, it is not clear to what extent mock classes are modified over time.

To better comprehend these factors, we assess the number of changes (*i.e.,* commits) on the mock classes. In addition to the number of changes, we also assess the number of distinct developers who perform them in the mock classes. We compare both the number of changes and developers that happen in mock classes with the same measures extracted from regular classes (*i.e.,* not mock classes).

We assess the statistical difference between mock and regular classes with the Mann-Whitney U Test [Mann and Whitney, 1947] ($\alpha = 0.05$). To assess the size of the difference, we compute the effect-size with the Cliff's Delta [Macbeth et al., 2011]. To assess code history and navigate on the commits, we rely on the PyDriller software repository mining tool [Spadini et al., 2018a].

3.4.4 RQ4 (Usage)

For the fourth research question, we focus on the client-side. We rely on the ultra-large-scale dataset Boa [Dyer et al., 2013], which includes over 2 million Java systems, to detect whether mock classes are used in the wild. In this case, our main goal is to examine external reuse of the mock classes. Specifically, we perform a query on all Java systems looking for import statements with the term "mock". We then filter out classes with the terms "Mockito", "Mockery", and "EasyMock" (*i.e.,* classes related to mocking frameworks), and also the ones that end with "Test" or "Tests" (*i.e.,* testing classes in Java). This way, we find 6,444 distinct classes that are imported 147,433 times. Table 3.4 summarizes the most frequent terms in these mock classes. As we can see, the most frequent terms are Service, Factory, and Context. Lastly, to answer our research question, we assess the frequency of our 604 mock classes in this dataset, as well as the most recurrent categories.

Table 3.4: Mock-related terms in Boa.

Pos	**Term**	#	**Pos**	**Term**	#
1	Service	411	6	Provider	161
2	Factory	299	7	Impl	148
3	Context	224	8	Request	145
4	Manager	202	9	Modules	138
5	Data	167	10	Test	136

3.4.5 RQ5 (Motivation) and RQ6 (Choice)

To answer RQ5 and RQ6, we rely on our survey with developers. In RQ5, we assess *why* developers rely on mock classes instead of mocking frameworks. For this purpose, we conduct a survey with developers who maintained mock classes. Specifically, after detecting that a developer modified a mock class, we ask *Why has a mocking framework (*e.g., *Mockito) not been used instead?* We aim to reveal possible usage cases and advantages of using mock classes, as perceived by the maintainer developer.

Lastly, in RQ6, we aim to assess *when* developers should create mock classes and mocking frameworks. To answer this research question, we assess the third question of the survey: *Based on your experience, when should developers create mock classes and when should they rely on mocking frameworks?* We aim to reveal whether developers perceive the cases that mock classes should be adopted and when mocking frameworks is a better solution.

3.5 Final Remarks

In this chapter, we presented the study design for this dissertation. We explained our strategy for selecting the software projects to be examined and our heuristic for detecting mock classes based on their names and usage of mocking frameworks. We also detailed our survey, providing its motivation and format. Finally, we presented the research questions we aim to answer with this work.

In the next two chapters, we present our empirical and survey results. Chapter 4 answers RQ1 to RQ4, while Chapter 5 details RQ5 and RQ6.

Chapter 4

Empirical Results

In this chapter, we present the empirical results obtained from the repository mining process. Section 4.1 presents what is the content of mock classes, by sorting them into categories (RQ1). Section 4.2 assesses the structure of the mock classes in terms of hierarchy and methods (RQ2). In Section 4.3, we present a study on class evolution (RQ3), and, in Section 4.4, the usage of the mock classes is examined via analysis of the Boa dataset (RQ4). Next, we present discussions and implications in Section 4.5, and, in Section 4.6, the threats to validity. Finally, Section 4.7 concludes the chapter.

4.1 RQ1 (Content): What is the content of mock classes?

Table 4.1 presents the categories of the mock classes detected after our manual classification. As we can see, the most common category is domain object (35%), followed by external dependency (23%) and web service (15%). Domain object (*i.e.,* classes that are mock to business rules) is present in 211 classes; examples mock classes in this category include: `MockAction` (Elasticsearch) and `MockCoreDescriptor` (Lucene-Solr). The second most frequent is external dependency (138 mock classes), which represents mock to external libraries; `AmazonEC2Mock` (Camel) and `MockGitHub` (Retrofit) are examples in this category. The third most frequent is web service (93 mock classes), that is, classes that are mock to web actions. Examples of web service include: `MockClientHttpResponse` (Spring Framework) and `MockHttpResource` (Elasticsearch). Other categories are Test support (11%, 67 mock classes), Native Java libraries (9%, 53 mock classes), and Network service (7%, 42 mock classes); notice that we do not find any mock class to the database category.

Table 4.1: Mock class categories.

Category	#	%
Domain object	211	35
External dependency	138	23
Web service	93	15
Test support	67	11
Native Java libraries	53	9
Network service	42	7
Database	0	0
Total	604	100

Previous studies report that domain objects, external dependencies, and web services are also among the most mocked categories when using mocking frameworks [Spadini et al., 2017, 2019] (*e.g.,* domain object is the most frequent in both our research and the mentioned studies). Another similarity with our results is regarding the least frequent categories: test support and native Java libraries are rarer in both analyses. Thus, our results complement the research literature by showing that, overall, developers tend to mock more and less frequently the same dependencies no matter they are mocking with the support of mocking frameworks or mock classes.

However, there are also some differences with regard to the results found in the mocking frameworks: we are not able to find mock classes in the database category, while in mocking frameworks this category is recurrent [Spadini et al., 2017, 2019]. We acknowledge that those differences regarding the database category may be because distinct projects on different domains are analyzed. Moreover, we are more strict in our definition of database category (*i.e.,* classes that are mock to SQL/NoSQL database library) to avoid subjectivity. Lastly, another difference is the network service category, which is present in our study and not in the ones about mocking frameworks [Spadini et al., 2017, 2019].

To complement this analysis, Table 4.2 presents the top-3 most frequent terms on each category. As we can notice, categories as web service and network service are dominated by terms that closely relate to their purposes. For instance, in the case of web service, the term *http* refers to the web protocol, *request* is a part of how the http protocol makes communication (request/response), and *mvc* is related to the architectural pattern Model-View-Controller. Similarly, the network service category includes terms as *channel*, *transport*, and *connection*, which are typical in the network domain. The categories domain object, native Java libraries, external dependency present somehow more generic related terms. This may be related to the fact that these categories are broader. Domain object is specific to each project, and those may be as varied as there are classes within the project. The same happens to native Java libraries and external dependencies, which are likely to be project-specific.

Summary RQ1: Mock classes are often created to emulate domain objects, external dependencies, and web services. Those categories are essentially the same types created by developers when using mocking frameworks [Spadini et al., 2017, 2019].

Table 4.2: Top-3 most frequent terms per mock class category.

Category	**Frequent Terms**
Domain object	Script, Filter, Factory
External dependency	Client, Amazon, Service
Web service	Http, Request, Mvc
Test support	Factory, Test, Mvc
Native Java libraries	Context, Config, Runnable
Network service	Channel, Transport, Connection

4.2 RQ2 (Design): How are mock classes designed?

We start by analyzing structural details of the mock classes, such as class extension and interface implementation. We then analyze the visibility of the mock classes and the number of methods in mock classes as compared to regular classes.

Class extension and interface implementation. Table 4.3 details the number of mock classes that are derived from class extension and interface implementation. Our first observation is that class extension is more frequent than interface implementation: 54.9% (332 out of 604) of the mock classes extend other classes, whereas 46.7% (282 out of 604) implement interfaces; only 7.5% (45 out of 604) do not extend classes nor implement interfaces. Notice, however, that this rate may vary per system: in Lucene-Solr, for instance, 82.6% of the mock classes are about extensions, while only 21.2% are interface implementation. On the other hand, in Dubbo, interface implementation is more common (82.8%) than class extension (17.2%).

Visibility. Table 4.4 summarizes the visibility of the studied mock classes. We can observe that the majority of the mock classes (68.4%, 413 out of 604) are *public*, thus, they are visible to all classes and can be reused. The *protected* visibility is the least frequent, presented in only 7 classes (1.2%). Next, 13.1% (79 out of 604) of

Table 4.3: Class extension and interface implementation in the mock classes.

Project	**Mock Classes**	**Class Extension**	%	**Interface Implementation**	%	**None**	%
Elasticsearch	138	88	63.8	56	40.6	13	9.4
Spring Boot	33	8	25.0	18	54.5	8	24.2
Guava	12	4	33.3	8	66.7	0	0
OkHttp	1	0	0.0	1	100.0	0	0
Spring Fw.	127	50	39.4	74	58.3	14	11.0
Retrofit	4	1	25.0	1	25.0	2	50.0
Dubbo	58	10	17.2	48	82.8	1	1.7
Lucene-Solr	132	109	82.6	28	21.2	2	1.5
Apache Camel	99	62	62.6	48	48.5	5	5.1
Total	604	332	54.9	282	46.7	45	7.5

Table 4.4: Visibility in the mock classes.

Project	**Mock Classes**	**Public**	%	**Protected**	%	**Package**	%	**Private**	%
Elasticsearch	138	84	60.1	5	3.7	17	12.3	32	23.2
Spring Boot	33	15	45.5	0	0	15	45.5	3	9.1
Guava	12	2	16.7	0	0	2	16.7	8	66.7
OkHttp	1	1	100.0	0	0	0	0	0	0
Spring Fw.	127	92	72.4	0	0	9	7.1	26	20.5
Retrofit	4	2	50.0	0	0	2	50.0	0	0
Dubbo	58	54	93.1	0	0	2	3.4	2	3.4
Lucene-Solr	132	77	58.3	2	1.5	24	18.2	29	21.9
Apache Camel	99	86	86.9	0	0	8	8.1	5	5.1
Total	604	413	68.4	7	1.2	79	13.1	105	17.4

the mock classes have the *package* visibility, that is, they are visible only within their own packages. Finally, 17.4% (105) of the mock classes are *private*, therefore, they are only accessed in their own classes. Interestingly, although reuse can be considered an important advantage for creating mock classes [Martin, 2014], the most restrictive controlling accesses (*package* and *private*) correspond to about 30% of the mock classes.

Number of methods. Table 4.5 presents some statistics about the number of methods in mock classes; for comparison, we also present the number of methods in randomly selected regular classes.[1] Although we find some variation among the investigated

[1]We randomly selected 604 classes from the target projects following the same distribution of the

Table 4.5: Number of methods in the mock and regular classes.

Project	**Mock Classes**			**Regular Classes**			**p-value**	$\neq$
	mean	**med**	σ	**mean**	**med**	σ		
Elasticsearch	4.6	2	5.1	5.5	4	6.3	0.07	-
Spring Boot	2.8	2	2.6	3.2	2	3.9	0.39	-
Guava	8	5.5	7.4	2.8	2	2.4	0.05	-
Spring Fw.	12.4	6	20.0	5.1	2	8.3	<0.05	medium
Dubbo	6.1	2.5	8.1	7.4	4.5	9.5	0.23	-
Lucene-Solr	3.7	2	5.6	5.3	3	5.4	<0.05	small
Apache Camel	10.9	4	21.0	4.9	3	6.8	0.06	-
All	7.2	3	13.8	5.3	3	6.9	0.07	-

projects, overall, both mock and regular classes have 3 methods on the median. Figure 4.1 contrasts the distribution of the number of methods in both groups. We can see in the figure that some mock classes are outliers, displaying a huge number of methods. Such a class is `AmazonIAMClientMock`,[2] from project Apache Camel, with 149 methods. About 90% of them only throw an exception informing that the operation is unsupported, which is trivial. By applying the Mann–Whitney test, we confirm both distributions of the number of methods is equivalent for all mock and regular classes (the difference is not statistically significant, with a *p-value*=0.07 compared to an α of 0.05). Project-wise, only two projects (Spring Framework and Lucene-Solr) had *p-values* $\leq$ 0.05. For these, effect size was computed via Cliff's Delta, with the interpretation of *medium* and *small* respectively.

Summary RQ2: Mock classes often extend other classes or implement interfaces. Around 70% of the mock classes are public and can be reused, while 30% have restrictive visibility (*i.e.,* private or package). Overall, mock and regular classes have the same number of methods, suggesting that mock classes are no simpler than regular classes in terms of structure and maintainability.

mock classes.

[2]`https://git.io/JuCMl`

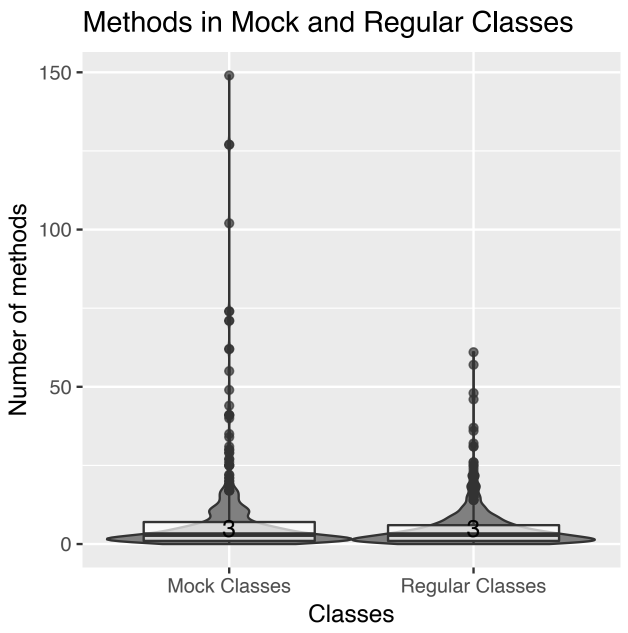


Figure 4.1: Distribution of the number of methods in mock and regular classes.

4.3 RQ3 (Evolution): How do mock classes evolve over time?

Table 4.6 summarizes the evolution of the mock classes in terms of the number of changes over time. For comparison purposes, we select a sample with the same number of regular classes. Overall, both mock and regular classes have 2 changes, on the median. By applying the Mann-Whitney U-test, we can see that both distributions are statistically equivalent (*p-value* $= 0.43$). When analyzing the changes per project, the distributions are mostly equivalent, with *p-value* ≥ 0.05 and negligible/small effect-size, meaning that the mock and regular classes have similar number of changes.

The sole exception is the Guava project: the mock classes have 1 change on the median, while the regular classes have 3.5 changes. In this case, the difference is statistically significant, with a large effect-size. This shows that the mock classes are stable in Guava and less likely to change than regular classes. We have inspected the mock classes provided by Guava and detected that they all happen inside test directories. This may suggest that those mock classes are intended to be used only within the project itself, and not by external clients. Notice, however, that the number of analyzed mock classes in Guava is smaller (only 12), as compared to the other systems.

In contrast, in the Spring Framework, the mock classes are likely to change, with 6 changes on the median. This is not very different from regular classes, which have 4 changes. However, it is worth to notice that the Spring Framework provides several mock classes as APIs.[3] In this case, the mock classes are intended to be used by external developers to facilitate web testing. Thus, one explanation is that mock classes in Spring Framework need to evolve as any other regular class to accommodate new features, fix-bugs, etc.

Table 4.6: Evolution of the classes - Changes.

		Mock classes			**Regular classes**				
Project	**#**	**mean**	**med**	σ	**mean**	**med**	σ	**p-value**	$\neq$
Elasticsearch	132	5.67	2.00	9.90	4.48	2.00	7.02	0.03	neg.
Spring Boot	31	2.00	2.00	1.26	2.81	1.00	3.23	0.28	-
Guava	12	1.75	1.00	1.29	6.42	3.50	6.91	0.01	lrg.
Spring Fw.	129	8.50	6.00	9.82	6.05	4.00	6.97	0.00	sml.
Dubbo	52	2.67	1.00	3.58	2.31	1.00	2.75	0.35	-
Lucene-Solr	132	4.89	2.50	9.62	6.28	3.00	8.15	0.04	neg.
Camel	98	2.15	2.00	1.75	4.56	2.00	7.98	0.03	sml.
All	586	4.99	2.00	8.42	5.00	2.00	7.13	0.43	-

Finally, to complement the prior analysis, overall, Table 4.7 shows that there is no significant difference between the number of unique developers that changed the mock classes and the same-sized sample of regular classes. Again, the sole difference happens in Guava.

Summary RQ3: Overall, mock classes and regular classes have equivalent number of changes and distinct developers. However, there may exist exceptions. For example, mock classes in Guava are more stable than regular classes, while in Spring Framework, which provide mock APIs, mock classes are likely to change, possibly to accommodate new features and bug-fixes.

4.4 RQ4 (Usage): How are mock classes used by developers?

In this research question, we focus on the client-side of the mock classes. For this purpose, we analyze the Boa dataset [Dyer et al., 2013] to assess mock classes. After

[3]https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html#mock-objects

Table 4.7: Evolution of the classes - Developers.

		Mock classes			Regular classes				
Project	#	**mean**	**med**	σ	**mean**	**med**	σ	**p-value**	$\neq$
Elasticsearch	132	2.96	2.00	3.18	2.44	1.00	2.11	0.11	-
Spring Boot	31	1.58	1.00	0.81	1.81	1.00	1.35	0.36	-
Guava	12	1.08	1.00	0.29	3.75	2.50	2.77	0.00	lrg.
Spring Fw.	129	4.80	4.00	3.24	3.64	3.00	2.56	0.00	sml.
Dubbo	52	1.69	1.00	1.65	1.71	1.00	1.47	0.33	-
Lucene-Solr	132	2.66	2.00	1.98	3.61	2.00	3.44	0.04	neg.
Apache Camel	98	1.72	1.50	0.96	2.81	2.00	3.03	0.01	sml.
All	586	2.87	2.00	2.68	2.96	2.00	2.74	0.38	-

filtering out possible false positives, we detect 6,444 mock classes that are used 147,433 times (see Section 3.2). This suggests that the adoption of mock classes is a common practice.

We now assess the 604 mock classes analyzed in this study to better understand how they are used in this large dataset by external clients. As presented in Table 4.8, we find that 128 out of the 604 mock classes are used 52,079 times. Web service is the most consumed category, being used by 24,022 clients (46%). Next, the second and third most consumed categories are domain object and external dependency, adopted by 11,879 (23%) and 7,624 (15%) clients, respectively. The least consumed categories are native Java library, test support, and network service.

Table 4.8: Most frequent mock categories in the Boa dataset.

Category	**#Classes**	**#Clients**	%	
Web service	35	24,022	46	▮
Domain object	39	11,879	23	▮
External dependency	14	7,624	15	▮
Native Java libraries	18	5,339	10	▮
Test support	7	1,899	4	▮
Network service	15	1,316	2	▮
Total	128	52,079	100	■

Figure 4.2 details the distribution of the usage per category. For example, on the median, the web service mock classes are used by 66 clients, while the domain object ones are used by 26 clients. The highest usage happens in native Java libraries (75 clients) and the lowest one happens in the category test support (10 clients). Interest-

ingly, although only 18 mock classes to native Java libraries are used in this dataset, they are highly consumed.

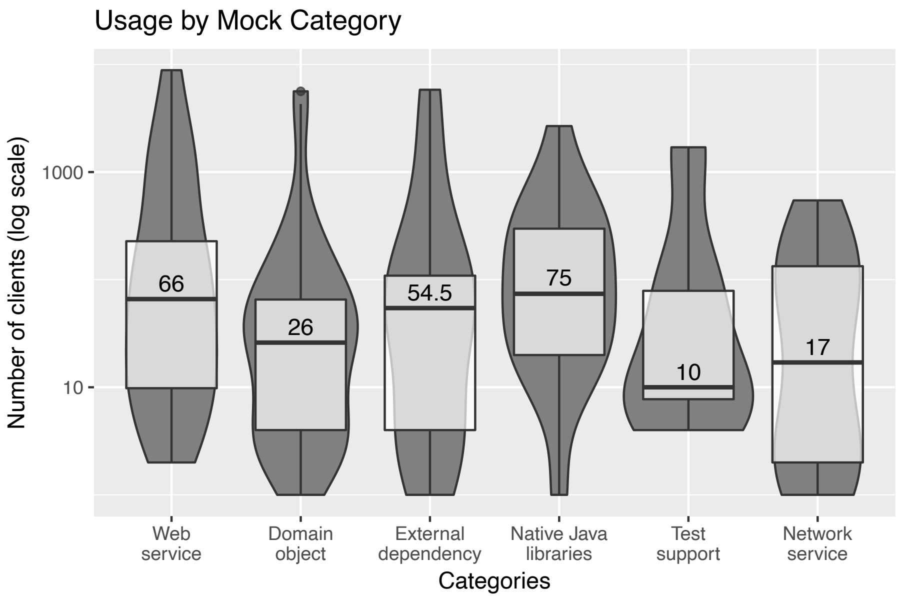


Figure 4.2: Usage of mock classes according to their categories.

Table 4.9 presents the top-10 most consumed mock classes. Classes `MockHttp-ServletRequest` and `MockHttpServletResponse` are the most used ones; they are both provided by Spring to facilitate web testing. The third one is `MockEndpoint`, which is provided by Apache Camel for testing routes and mediation rules using mocks. Half of the top-10 most consumed classes are about web services. Together, the top-10 mock classes are used by 39,693 out of 52,079 clients (76%), while the top-50 are responsible for almost all client usage (97%).

Summary RQ4: Mock classes are largely consumed by client projects to support testing; web services are the most emulated dependencies. The usage is very concentrated on a few classes: 10 classes are used by 76% of the clients, while 50 classes are consumed by 97%.

4.5 Discussion and Implications

4.5.1 Novel empirical data on mock classes

Mock objects are often used to support software testing [Feathers, 2004; Meszaros, 2007; Freeman and Pryce, 2009; Osherove, 2009; Percival, 2014], however, there is a surprising

Table 4.9: Most frequent mock classes in the Boa dataset.

Mock Class	**Mock Category**	#
MockHttpServletRequest	Web service	8,846
MockHttpServletResponse	Web service	6,104
MockEndpoint	External dependency	5,821
MockAnalyzer	Domain object	5,628
MockTokenizer	Domain object	4,288
MockServletContext	Native Java libraries	2,675
MockMvc	Web service	1,822
MockContext	Test support	1,699
MockResponse	Web service	1,575
MockWebServer	Web service	1,235
Top-10	-	39,693
Top-50	-	50,709

lack of research studies on that topic. Although mock classes are typically provided by large and popular projects (as the ones assessed in this research, *e.g.,* Elasticsearch, Spring Framework, and Lucene-Solr), actual mock studies are limited to the context of mocking frameworks [Mostafa and Wang, 2014; Arcuri et al., 2017; Spadini et al., 2017, 2019]. *Therefore, we contribute to the software testing literature with a novel study about mock classes and their usage in order to complement existing research in the context of mocking frameworks.*

Moreover, our study reveals new data about mock classes. In RQ1, we find that mock classes are concentrated on the categories domain object, external dependency, and web service (which is similar to the mocks created by frameworks). We also present several structural information about the mock classes (RQ2). For example, they mostly rely on inheritance and interface implementation (that is, mock classes are rarely standalone) and they are typically public and can be reused but private mock classes are not rare. Furthermore, mock classes and regular classes have equivalent number of changes, meaning that both classes demand similar effort to maintain (RQ3). *Thus, we reveal novel quantitative and qualitative empirical data about the creation of mock classes, which can guide practitioners in charge of maintaining them. We show that mock classes are over-concentrated on certain tasks, are often part of a hierarchy, and are mostly public. Also, mock classes are not different from regular classes regarding number of methods and number of changes.*

4.5.2 Reuse and lack of visibility of the mock classes

One of the benefits of creating mock classes by hand is their reuse power [Martin, 2014]. For example, a single mock class provided by system X can be used to support the creation of test cases in X itself and in the clients of X. Indeed, as we presented in RQ2, mock classes may have thousands of clients. However, we find that about 30% of the analyzed mock classes have *package* or *private* visibility. That is, their reuse is very limited: they are visible only within their own packages or classes. Thus, the lack of reusability on almost one-third of the mock classes is a surprising result. As those private mock classes are not intended to be reused, they are straightforward candidates to be mocked with frameworks; in this case, the maintenance effort would be smaller as less mock classes would be available. *Thus, we shed light on the over creation of private mock classes, which may be harmful to the overall project maintainability. This can drive future research agenda on techniques to detect superfluous mock classes that can be created with mocking frameworks.*

4.5.3 Widespread usage of the mock classes

We also find that mock classes are used to a larger extent (RQ4). When analyzing the Boa dataset [Dyer et al., 2013], we detected that mock classes are consumed by thousands of open-source software projects. That is, the usage of mock classes is not restricted to the target projects of this study, but it seems to be widespread. *Thus, practitioners who maintain mock classes (as the ones assessed in this research) should be aware of the importance of these classes to their ecosystems. During maintenance, mock classes should be changed with care because a large number of client projects can be impacted.*

4.6 Threats to validity

4.6.1 Focus on libraries instead of end-user products

The projects we selected in this study are libraries or frameworks that are typically used as dependencies by end-user software. For example, Spring Boot is a framework used to build web applications, and not a final application itself. Indeed, libraries and frameworks are really important to support software development nowadays, providing feature reuse, improving productivity, and decreasing costs [Moser and Nierstrasz, 1996; Konstantopoulos et al., 2009; Raemaekers et al., 2012; Menezes et al., 2019; Lima and Hora, 2020]. However, we recognize they do not comprehend the reality of those end-user products when using mock classes, so this needs to be taken into account when interpreting our categories of mock classes. Notice that end-user software products are better represented in the results presented in RQ3, in which we assessed millions of Java projects with the support of the Boa platform [Dyer et al., 2013]. In this case, web services were the most common mocked dependencies.

4.6.2 Manual classification

Mock classes in our study were manually classified by the author, who is a software engineer with 8 years of experience in embedded, desktop, and web development. Many of them were classified based on strong terms present in their names (*e.g.,* Mvc, Http, Rest, TCP, GitHub, etc.). In cases the names were not clear (or in cases of doubts), the author relied on additional artifacts, and carefully analyzed documentation or inspected the mock source code to infer the category (see Chapter 3 for more details). Thus, like any other manual classification, it is subjected to error and bias. However, an evidence that may minimize this threat is that the frequency of mock categories detected in this study is similar to those found by earlier studies on mocking frameworks [Spadini et al., 2017, 2019], that is, domain objects, web services, and external dependencies are frequent on both research studies, while test support and native Java libraries are less common.

4.6.3 Lack of mock classes categorized as Database

In a previous study about mock objects and mocking frameworks in Java [Spadini et al., 2017, 2019], a database mock is defined as one "*that interact with an external database. These classes can be either an external library [...] or a class that depends on such external libraries [...]*". In our study, we were more strict in the definition of database: we only stated a mock class to be in the database category when it was directly linked to a SQL/NoSQL database library. We were more strict due to two reasons. First, we found the original definition a bit subjective and flexible. Second, one of the projects in our study, Elasticsearch, is itself a NoSQL database, thus, to some extent, all mock classes in this system could be classified in the database category, and, of course, this would not be desirable. Therefore, according to our criteria, we found no mock classes for databases.

4.6.4 Identifying mock classes

Our selection criterion for identifying mock classes is that they should contain the string "mock" in the class name. While our findings show that there is plenty of mock classes that follow this guideline, there is the possibility we lose track of classes that are used as mocks but do not follow it.

4.6.5 Generalization

We analyzed 604 mock classes provided by several popular and real-world Java software systems. For instance, the projects Elasticsearch, Spring Boot, and RxJava have over 40k stars, thus, they are among the most popular in the Java ecosystem, as measured by Github. Moreover, in our third research question, we searched for the usage of mocks in millions of projects with the Boa platform [Dyer et al., 2013]. Despite these observations, our findings — as usual in empirical software engineering — may not be directly generalized to other systems, as commercial ones with closed source and implemented in other programming languages.

4.7 Final Remarks

In this chapter, we detailed the empirical results of this dissertation. By using them to answer the first four research questions, we shed light on mock classes usage. In RQ1, we showed that mock classes are often created to emulate domain objects, external dependencies, and web services. In RQ2, we presented that mock classes often extend other classes or implement interfaces, and overall have the same number of methods of the regular classes. RQ3 presented that mock classes usually have a number of changes and distinct developers equivalent to regular classes, but exceptions exist. Finally, the answer to RQ4 was that mock classes are largely consumed by client projects to support testing, with usage being concentrated on a few classes.

The next chapter complements the quantitative study of this chapter, with a qualitative assessment conducted in a survey with expert developers.

Chapter 5

Survey Results

In this chapter, we present our survey results. Before we look at the research questions, answered by examining Questions 2 and 3 of the survey, we check the survey's Question 1 to understand what was the goal of the discovered mock classes as stated by those who have worked with them. Section 5.1 assesses why mock classes are used by developers and Section 5.2 focus on when mock classes should be adopted. We present a discussion of the findings (Section 5.3) and threats to validity (Section 5.4). Lastly, Section 5.5 presents the final remarks.

Q1. What is the goal of this mock class?

Table 5.1 summarizes the goal of the investigated mock classes, as reported by the developers. ***Simulate entity*** is the most common goal for the mock classes (49%). For example, Developer #11 says: "*The class is used to mock a TableSource to test the Table Schema and properties of the table source.*". Developer #8 mentions: "*This is a mock 'SourceReader'/'Source' implementation to be used in tests. For example if another component that we want to test requires 'SourceReader' as a dependency (...).*". Likewise, Developer #7 states that the class in question helps on "*Allowing to test the interaction between the code under test and some service/other component.*".

The second most frequent goal is ***mock external service*** (17%). This category relates to the simulation of any external service, such as database, application servers, and APIs. In this context, Developer #21 comments that the class is used "*[t]o provide a mock of an Rpc interface for testing*".

Next, we have the category ***interface implementation*** (11%). In this case, developers wanted to use the interface implementation in a test, so they created a mock class for the purpose of implementing that interface. For example, Developer #26 states plainly that they want "*[t]o implement the HDFSWriter interface for our*

Table 5.1: Categories for Question 1: What is the goal of this mock class?

Category	**# of answers**	**% of answers**
Simulate entity	17	49
Mock external service	6	17
Interface implementation	4	11
Mock environment	2	6
Others	6	17
Total	35	100

test". Note that Mockito provides a functionality for instantiating an object that respects an interface, without the need for creating a class that implements it.[1]

Another category is ***mock environment*** (6%). In this case, developers aimed to simulate environmental conditions, such as system time and memory handling. Developer #28 mentions the need for "*(allowing) you to implement inmemory metadata for testing*". Finally, the category **others** is composed of answers for very specific needs, like tracing requests: "*I created this class to just trace the received http requests*" (Developer #9).

Summary Q1: Mock classes are mainly created to simulate entities. This goal is not different from developers who rely on mocking frameworks.

5.1 RQ5 (Motivation): Why do developers rely on mock classes instead of mocking frameworks?

Next, we study why mocking frameworks have not been adopted instead of mock classes. For this, we look into the answers of the question 2 on the survey, *Why has a mocking framework not been used instead?* Table 5.2 summarizes this analysis: the answers are divided in two groups according to the position of the developers on the mocking practice. *Against* represents answers in which developers are negative about the usage of mocking frameworks, while *in favor* presents answers in which they are positive about the usage of mock classes.

[1]The Mockito `mock` API *"creates mock object of given class or interface"* (`https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html`).

Table 5.2: Categories for Question 2: Why has a mocking framework not been used instead?

Type	Category	# of answers	% of answers
Against	Mocking frameworks reduce code quality	7	41
	Mocking frameworks have limitations	6	35
	Mocking frameworks add dependencies	4	24
	Total	17	100
In favor	Mock classes improve code quality	4	36
	Usage of local mocking framework	4	36
	Mock classes are simple to set up	3	28
	Total	11	100
	Others	6	43
	Does not know	5	36
	Mocking framework is unavailable	3	21
	Total	14	100

5.1.1 Against Mocking Frameworks

We find three categories against mocking frameworks. The most frequent is ***mocking frameworks reduce code quality*** (41%), in which developers present concerns regarding code maintenance, testability, and understandability. Developer #7 states: "*sing a mock library too extensively makes maintenance harder because one has to touch many places if the mocked class changes*". Likewise, Developer #21 mentions: "*Mocking frameworks (...) make it hard to ascertain if the configuration of classes under test behave well under the regular rules of the language [and] removes the force towards better designs normally caused by requiring testability*". Developer #28 says: "*(...) the mocks generated by a mocking framework are hard to maintain and understand*".

Developers also mention ***mocking frameworks have limitations*** as a reason against their usage (35%). For example, Developer #33 states: "*A simple single-class mock does not suffice here [mocking the networking layer of the system]*". Another problem related by developers is ***mocking frameworks add dependencies*** (24%). Developer #32 mentions: "*(...) adding a bunch of new dependencies with your commit does not make the reviewers happier. Deciding which mocking framework to use, and when, is a team decision*".

5.1.2 In Favor of Mock Classes

On the other hand, there are developers in favor of the usage of mock classes (rather than against mocking frameworks). They argue that ***mock classes improve code quality***. For example, Developer #29 answers: "*It's clearer and more readable to contain the mock dependencies in a well defined regular class, so you have an implementation to turn to*". ***Usage of local mocking framework*** represents cases that the project itself provides a mocking framework that can be used by developers. For example, Developer #39 mentions: "*[the project] has its own built-in mocking framework*". Developers also mention that ***mock classes are simple to set up***. In this context, Developer #24 states: "*[The mock class] is better than using Mockito in this case because it is used for large tests that have lots of interactions with Curator*".

Summary RQ5: Developers rely on mock classes over mocking frameworks due to several reasons. They argue that mocking frameworks reduce code quality, have limitations, and add dependencies. Also, developers mention that mock classes improve code quality and are simple to set up in tests.

5.2 RQ6 (Choice): When should developers create mock classes and when is it better to rely on mocking frameworks?

The last survey question (*When should developers create mock classes and when should they rely on mocking frameworks?*) assess when developers should create mock classes and when a mocking framework is a better choice. As summarized in Table 5.3, we group the answers for the *mocking frameworks* and for the *mock classes.*

5.2.1 When developers should create mock classes

First, we assess when developers should create mock classes. The most frequent explanation is ***for complex testing***. In this case, they are stating that mock classes are a better way to deal with tests that are larger or more complex. For example, Developer #33 states: "*when mocking a multitude of classes which constitute a large, integral part of the system, it's better to do it manually*". Developer #35 states that mock classes could be used on "*cases where you want to simulate a complex system*", but pointing out that possibly "*there are testing frameworks dedicated to these complex systems*". In

Table 5.3: Categories for Question 3: When should developers create mock classes and when should they rely on mocking frameworks?

Type	Category	# of answers	% of answers
Mock classes	For complex testing	5	24
	To avoid mocking framework limitations	4	19
	To have better code quality	4	19
	To support reuse	4	19
	To keep project quality	4	19
	Total	21	100
Mocking frameworks	For simple unit tests	6	60
	To test external services	3	30
	To test code with minimal change	1	10
	Total	10	100
	Generally, use mocking frameworks	10	53
	Avoid using mocks	6	31
	Avoid using mocking frameworks	3	16
	Total	19	100

this context, Developer #37 specifically declares that it is preferable to use "*custom mocks if the class being mocked needs complex behavior or needs to keep state*".

All the following categories had the same number of answers (*i.e.,* 4). In the first one, developers state about that they should use mock classes when they want ***to avoid mocking framework limitations***. For example, Developer #1 answers: "*The java mock frameworks are usually pretty badly behaved, (...) need too many security permissions*". More directly, Developer #39 mentions: "*a home-brew mock class is helpful if using [a mocking framework] does not provide the features needed*".

Another explanation is ***to have better code quality***, exemplified by Developer #28: "*(...) I'm not a fan of mocking frameworks. I prefer to better design the application, so the testing process doesn't require tricky mocks that can be achieved only by applying a mocking framework*". Likewise, Developer #39 answers:"*using [mocking frameworks] would result in contrived and difficult to read and maintain code*". Another scenario happens ***to support reuse***. Developer #4 answers that it is useful when "*you want to provide other developers or other teams with a better way of testing more code with less efforts that are specific to [your product]*". On a similar line, Developer #43 says: "*reason for preferring mock classes is to document usage of an API, simulating what a user has to do*".

A last category for when mock classes are preferable is ***to keep project quality***. These answers are different from those that focus specifically on code-quality aspects by being concerned with the quality of the project as a whole. Developer #3 mentions: "*When this is a single use in a project that has hundreds of modules and dependencies, adding a new dependency might be an overkill*". Developer #25 is concerned with the project performance: "*reason not to use a framework is performance (...) [We had] a case where a very simple mock from a framework degraded performance by a factor of 1000*".

5.2.2 When developers should rely on mocking frameworks

We now assess when developers should rely on mocking frameworks. The category with most answers is ***for simple unit tests.***. Developer #24 states that they "*prefer mocking frameworks for (small) unit testing, and mock classes for everything else*". Likewise, Developer #33 mentions: "*use mocking tools in our unit tests when mocking a small, isolated class*". Interestingly, this category contrasts with the first one presented for the mock classes, which is *for complex testing.* According to the surveyed developers, while mock classes seem to be used for complex tests, mocking frameworks is adopted for simple ones.

The next category is ***to test external services***. Developer #18 mentions: "*I would use Mockito when there is an external service, or there is need to start a server in the framework, Mockito can help to write UTs instead of ITs*".[2]

The last category on when developers should rely on mocking frameworks is ***to test code with minimal change***. In this regard, developer #26 says: "*Mockito has the advantage that it can test code with minimal change. For a legacy system where refactoring is too expensive it is a good tool*".

Finally, some answers are more generic and do not fit in any category. Some developers just say that in general they would simply use mocking frameworks. For example, Developer #4 mentions: "*sticking to popular frameworks will be more productive*", while Developer #16 goes even further by stating that they "*would normally use a mocking framework, usually Mockito, when writing a library myself. I don't know of a concrete example of when a mocking class is preferred*". Other developers are against mock practices in general and advise to avoid them. For example, Developer #10 states: "*The only suggestion I have regarding mocks is: please don't use them*". Some developers are against mocking frameworks in particular. Developer #7 answers: "*We*

[2]UTs: Unit Tests, ITs: Integration Tests

try not [to] use Mockito wherever possible", while Developer #19 states: "*We generally avoid using Mockito in [project]*".

Summary RQ6: According to the surveyed developers, there are many reasons to create mock classes: for complex testing, to avoid mocking framework limitations, to have better code quality, to support reuse, and to keep project quality. On the other hand, developers should rely on mocking frameworks mainly to create simple unit tests, but also to test external entities and to test code with minimal change.

5.3 Discussion and Implications

5.3.1 Reasons for Using Mock Classes and Mocking Frameworks

Overall, the frequency of mock categories detected in this study is similar to those found by earlier studies on mocking frameworks [Spadini et al., 2017, 2019]. For example, domain objects, web services, and external dependencies are common in both studies, while test support and native Java libraries are rarer (RQ1). This suggests that independently of the way developers are mocking dependencies (*i.e.,* either via mock classes or mocking frameworks), the overall goal is the same. Indeed, as presented in Table 3.1, most of the analyzed projects in this study use both solutions to mock dependencies. Thus, one question arises: *why* and *when* do developers rely on mock classes instead of mocking frameworks? We aimed to address this question in our survey analysis (RQ5 and RQ6) and we shed some light on that direction:

- *Why.* Developers state that mock classes improve code quality and are simple to set up in tests. In contrast, developers mention that mocking frameworks reduce code quality, have limitations, and add dependencies.
- *When.* Developers state several reasons to create mock classes: for complex testing, to avoid mocking framework limitations, to have better code quality, to support reuse, and to keep project quality. In contrast, developers should rely on mocking frameworks mainly to create simple unit tests, but also to test external entities and to test code with minimal change.

Developers are pragmatic when selecting between mock classes and mocking frameworks. *The overall perception is that the usage of mock classes is preferable over* *mocking frameworks, however, there are specific cases in which mocking frameworks are a better choice. We thus shed light on the benefits of mock classes instead of mocking frameworks (and vice-versa). This information can guide practitioners when choosing the ideal solution to emulate dependencies in tests.*

5.3.2 Mock and test double terminology

One of the challenges that happened during the survey study was related to what developers consider to be a mock. In Chapter 2, we presented that the state of the practice is to frequently use the terms mocks and test doubles interchangeably. While a minority of the surveyed developers used terms related to Meszaros' taxonomy on test doubles [Meszaros, 2007; Fowler, 2007] such as *stub* or *spy*, the majority were comfortable with the plain use of "mock". Another point commented by the surveyed developers was that custom mock classes may be perceived as (local) mocking frameworks. For example, Developer #4 mentions: "*The mock class is a part of mocking framework mock-mvc* [3] *that lies within spring-test library. (...) So using this particular library allows one to test bigger chunks of infrastructure (e.g json de-/serialization) while also providing a nice language that is specific to HTTP requests and responses, comparing to what popular mocking frameworks provide - a language designed to create mocks*". Since those classes are not dedicated tools, but a collection of mock classes to facilitate the creation of tests, we did not consider them as mocking frameworks. Nevertheless, it is interesting to note that developers may consider a collection of mock classes as a kind of mocking framework. *In this study, we shed some light on the mock terminologies. We reinforce "mock" as a well-known term adopted for test doubles, confirming the widespread usage in practice. Moreover, a collection of custom mock classes may be perceived as (local) mocking frameworks. We thus recommend that studies on custom mock classes and mocking frameworks make clear the differences.*

5.4 Threats to Validity

5.4.1 Survey answers and assessment

Initially, we looked for the developers of changes in the mock classes in the original 12 projects. This approach, however, generated a low number of candidate developers and answers. To find more candidate developers, we looked for mock classes in other 63 popular Java projects. To minimize subjectivity, the survey analysis has been performed

[3]`https://git.io/JR1m8`

with special attention by the author and the advisor via *thematic analysis* [Cruzes and Dyba, 2011].

5.4.2 Terminology and context

The usage of the “mock” terminology may not be equally interpreted among respondent developers. For example, some developers demonstrated confusion on this terminology, mixing local mocking frameworks and mock classes. Moreover, different project contexts could have affected developers’ perceptions on mock usage and necessity, *e.g.,* when a project was old enough so it was developed prior to mocking frameworks widespread usage and lacked the interest for moving to mocking frameworks.

5.5 Final Remarks

In this chapter, we presented the data analysis of the survey responses. We used this analysis to answer to our last two research questions: what are the reasons for using mock classes instead of mocking frameworks, and when should developers create mock classes or rely on mocking frameworks. We found that developers have these preferences due to several reasons. They argue that mocking frameworks reduce code quality, have limitations, and add dependencies. On the other hand, they mention that mock classes improve code quality and are simple to use in tests. Developers mention that mock classes can be created for complex testing and to avoid the aforementioned mocking frameworks limitations. Developers also cite code quality concerns when describing reasons to create mock classes. They prefer to rely on mocking frameworks to create simple unit tests, test external entities, and to test code with minimal change.

Chapter 6

Conclusion

6.1 Summary and Contributions

During testing activities, developers frequently rely on dependencies that make the test harder to be implemented. In this case, developers can use mock objects to emulate the dependencies' behavior, which contributes to make the test fast and isolated. The emulated dependency can be dynamically created with the support of mocking frameworks or manually hand-coded in *mock classes* [Meszaros, 2007]. Past research showed that mocking frameworks are largely adopted by software projects [Mostafa and Wang, 2014] and that they may indeed support the creation of unit tests [Marri et al., 2009; Arcuri et al., 2017; Spadini et al., 2017, 2019]. However, those researches are restricted to the context of mocking frameworks.

In this dissertation, we presented an empirical and survey study to assess mock classes. We analyzed 12 popular Java projects and detected 604 mock classes. We summarize our empirical findings as follows: mock classes are often created to emulate domain objects, external dependencies, and web services (RQ1); mock classes are often part of a hierarchy, are mostly public, and are not different from regular classes regarding number of methods (RQ2); mock classes and regular classes have equivalent number of changes (RQ3); and mock classes are largely consumed by client projects to support testing, particularly to emulate web services (RQ4).

Our survey study with 39 developers reveals the the following findings. Mock classes improve code quality and are simple to set up in tests. In contrast, developers mention that mocking frameworks reduce code quality, have limitations, and add dependencies (RQ5). Mock classes should be created for complex testing, to avoid mocking framework limitations, to have better code quality, to support reuse, and to keep project quality. In contrast, developers should rely on mocking frameworks mainly to create simple unit tests, but also to test external entities and to test code with minimal change (RQ6).

Based on our results, we provided insights and practical implications for researchers and practitioners by discussing topics as (i) the novel empirical data on mock classes, (ii) the reusability and lack of visibility of the mock classes, (iii) the widespread usage of the mock classes, and (iv) the reasons for using mock classes and mocking frameworks.

6.2 Future Work

As future work, we plan to take a deeper look at the semantics of the mock classes to refine their classification and provide insights on how they mock dependencies. We plan to look for mock classes in other programming languages (*e.g.,* Python and JavaScript) to assess the concerns of other software ecosystems. We also plan to study the implementation techniques used for developing the mocking frameworks. This would help on understanding the problems and limitations pointed by developers in the survey regarding the usage of such frameworks. In the present work, we examined external reuse of the mock classes. We did not examine, however, the internal reuse of mock class. Thus, this study would help to better examine questions such as visibility and class hierarchy. Code duplication as a bad practice related to reuse is also an interesting possibility for further investigations. The systems we studied are large open-source projects, and may not represent the experience of most developers that work on smaller closed-source software projects. Therefore, assessing proprietary systems would be useful to understand topics such as the mocking of database systems, for example, and the overall use of mock classes vs. mocking frameworks in such contexts. Finally, another interesting research direction is to automatically suggest mocking framework usage based on the mock classes that already exist, as recently proposed by Wang et al. [2021].

Bibliography

Arcuri, A., Fraser, G., and Just, R. (2017). Private API access and functional mocking in automated unit test generation. In *International Conference on Software Testing, Verification and Validation*, pages 126--137.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2015). Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052--1094.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85--103. IEEE.

Bhagya, T., Dietrich, J., and Guesgen, H. (2019). Generating mock skeletons for lightweight web-service testing. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 181--188. IEEE.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *International Conference on Software Maintenance and Evolution*, pages 334--344.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). Pattern-oriented software architecture volume 1: A system of patterns.

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional.

Cruzes, D. S. and Dyba, T. (2011). Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering*, pages 422--431.

EasyMock (2021). Easymock: Easy mocking. better testing. *https://easymock.org/*.

Elliott, E. (2017). Mocking is a code smell. *https://medium.com/javascript-scene/mocking-is-a-code-smell-944a70c90a6a*. Library Catalog: medium.com.

Facebook (2021). Jest testing framework. *https://jestjs.io/*.

Feathers, M. (2004). *Working Effectively with Legacy Code*. Prentice Hall Professional.

Fowler, M. (2006). bliki: TestDouble. *https://martinfowler.com/bliki/TestDouble.html*.

Fowler, M. (2007). Mocks aren't stubs. *https://martinfowler.com/articles/mocksArentStubs.html*.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Freeman, S., Mackinnon, T., Pryce, N., and Walnes, J. (2004). Mock roles, not objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236--246.

Freeman, S. and Pryce, N. (2009). *Growing object-oriented software, guided by tests*. Pearson Education.

jMock (2007). jmock. *http://jmock.org/*.

Johnson, R. E. (1992). Documenting frameworks using patterns. In *conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63--76.

Khorikov, V. (2020). *Unit Testing Principles, Practices, and Patterns*. Manning Publications.

Konstantopoulos, D., Marien, J., Pinkerton, M., and Braude, E. (2009). Best principles in the design of shared software. In *International Computer Software and Applications Conference*, pages 287–292.

Lima, C. and Hora, A. (2020). What are the characteristics of popular apis? a large scale study on java, android, and 165 libraries. In *Software Quality Journal*, volume 28, page 425–458.

Macbeth, G., Razumiejczyk, E., and Ledesma, R. D. (2011). Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545--555.

Mackinnon, T., Freeman, S., and Craig, P. (2000). Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287--301.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50--60.

Marri, M. R., Xie, T., Tillmann, N., De Halleux, J., and Schulte, W. (2009). An empirical study of testing file-system-dependent software with mock objects. In *Workshop on Automation of Software Test*, pages 149--153. IEEE.

Martin, R. (2014). When to mock - the clean code blog. *https://blog.cleancoder.com/uncle-bob/2014/05/10/WhenToMock.html*.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Menezes, G., Cafeo, B., and Hora, A. (2019). Framework code samples: How are they maintained and used by developers? In *13th International Symposium on Empirical Software Engineering and Measurement*, pages 1--11.

Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Pearson Education.

Mockito (2021). Mockito: Tasty mocking framework for unit tests in java. *https://site.mockito.org/*.

moq (2021). moq - the most popular and friendly mocking library for .net. *https://github.com/moq/moq4*.

Moser, S. and Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9).

Mostafa, S. and Wang, X. (2014). An empirical study on the usage of mocking frameworks in software testing. In *International Conference on Quality Software*, pages 127--132.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

Osherove, R. (2009). *The Art of Unit Testing: With Examples in. Net*. Manning Publications Co.

Palomba, F. and Zaidman, A. (2017). Does refactoring of test smells induce fixing flaky tests? In *International Conference on Software Maintenance and Evolution*, pages 1--12.

Percival, H. (2014). *Test-driven development with Python: obey the testing goat: using Django, Selenium, and JavaScript.* O'Reilly Media, Inc.

Python Software Foundation (2020). unittest.mock — mock object library — python 3.8.2 documentation. *https://docs.python.org/3/library/unittest.mock.html.*

Raemaekers, S., van Deursen, A., and Visser, J. (2012). Measuring software library stability through historical version analysis. In *International Conference on Software Maintenance*, pages 378–387.

Silva, H. and Valente, M. T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112--129.

Sinon.js (2021). Sinon.js. *https://sinonjs.org.*

Sivashanmugam, K. and Palanisamy, S. (2009). Testing sql server integration services runtime engine using model and mock objects. In *2009 Sixth International Conference on Information Technology: New Generations*, pages 790--795. IEEE.

Spadini, D., Aniche, M., and Bacchelli, A. (2018a). Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908--911.

Spadini, D., Aniche, M., Bruntink, M., and Bacchelli, A. (2017). To mock or not to mock? an empirical study on mocking practices. In *International Conference on Mining Software Repositories*, pages 402--412.

Spadini, D., Aniche, M., Bruntink, M., and Bacchelli, A. (2019). Mock objects for testing java systems. *Empirical Software Engineering*, 24:1461--1498.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. (2018b). On the relation of test smells to software code quality. In *International Conference on Software Maintenance and Evolution*, pages 1--12.

Thomas, D. and Hunt, A. (2002). Mock objects. *IEEE Software*, 19(3):22--24.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *International Conference on Automated Software Engineering*, pages 4--15.

Valente, M. T. (2020). Engenharia de software moderna (livro digital).

van Bladel, B. and Demeyer, S. (2020). Clone detection in test code: An empirical evaluation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 492–500.

Vocke, H. (2018). The practical test pyramid. *https://martinfowler.com/articles/practical-test-pyramid.html.*

Wang, X., Xiao, L., Yu, T., Woepse, A., and Wong, S. (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*, pages 540--552.

Winters, T., Manshreck, T., and Wright, H. (2020). *Software engineering at google: Lessons learned from programming over time.* O'Reilly Media.

Zaidman, A., Van Rompaey, B., van Deursen, A., and Demeyer, S. (2011). Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325--364.