Use of Test Doubles in Android Testing: An In-Depth Investigation

Mattia Fazzini*, Chase Choi*, Juan Manuel Copia†, Gabriel Lee*,
Yoshiki Kakehi‡, Alessandra Gorla†, Alessandro Orso‡
*University of Minnesota, Minneapolis, MN, USA; mfazzini@umn.edu, choix698@umn.edu, gnlee@umn.edu
†IMDEA Software Institute, Madrid, Spain; juanmanuel.copia@imdea.org, alessandra.gorla@imdea.org
‡Georgia Institute of Technology, Atlanta, GA, USA; yoshikikakehi@gatech.edu, orso@cc.gatech.edu

ABSTRACT

Android apps interact with their environment extensively, which can result in flaky, slow, or hard-to-debug tests. Developers often address these problems using test doubles—developer-defined objects that replace app or library classes during test execution. Although test doubles are widely used, there is limited understanding of how they are used in practice. To bridge this gap, we present an in-depth empirical study that aims to shed light on how developers create and use test doubles in Android apps. In our study, we first analyze a dataset of 1,006 apps with publicly available test suites to identify which frameworks and approaches developers most commonly use to create test doubles. We then investigate several research questions by studying how test doubles defined using these popular frameworks are created and used in the ten apps in the dataset that define the highest number of test doubles using these frameworks. Our results, based on the analysis of 2,365 test doubles that replace a total of 784 classes, provide insight into the types of test doubles used within Android apps and how they are utilized. Our results also show that test doubles used in Android apps and traditional Java test doubles differ in at least some respect. Finally, our results show that test doubles can introduce test smells and even mistakes in the test code. In the paper, we also discuss some implications of our findings that can help researchers and practitioners working in this area and guide future research.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Test mocking, mobile apps, software environment

ACM Reference Format:
Mattia Fazzini*, Chase Choi*, Juan Manuel Copia†, Gabriel Lee*,, Yoshiki Kakehi‡, Alessandra Gorla†, Alessandro Orso‡. 2022. Use of Test Doubles in Android Testing: An In-Depth Investigation. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510175

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

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510175

1 INTRODUCTION

Most Android apps have rich interactions with their environment. Android devices, for instance, provide built-in motion, location information, and position sensors that apps can use to offer a rich set of features to users. In general, apps interface with external web services, the underlying Android system, third-party libraries, as well as content providers exposed by the device. Such extensive interactions complicate the testing of an app, as exploring specific behaviors may require complex configurations of the environment, and test execution may become slow and result in flakiness.

To mitigate these issues, developers can rely on *test doubles (TDs)*—classes that mimic the structure of other classes but offer alternative implementations that are fully controlled by the developer for the purpose of testing. In the context of Android apps, TDs can replace classes defined in the app itself, classes from the Java library, classes defined in third-party libraries, and classes from the Android framework. Furthermore, depending on their purpose, TDs may be classified as follows: (i) *dummies*, which are often used to simply fill-in parameters that are meaningless for a specific test; (ii) *stubs*, which are simple objects that return hard-coded values when their methods are invoked; (iii) *mocks* and *spies*, which are more complex objects that can verify interactions with other classes; and (iv) *fakes*, which consist of partially working implementations that are more efficient than the actual class(es) they are replacing.

Because creating and maintaining TDs can involve considerable manual effort, researchers have started investigating techniques to support developers in this task (e.g., [1–5]). Unfortunately, however, there is limited understanding of how TDs are used in practice, which hinders our ability to define effective techniques in this space. Several previous empirical studies aimed to identify general testing practices in the development of Android apps [6–11], but they either ignored or did not specifically focus on TDs. Other related studies analyzed how Java developers use mocks when testing traditional (i.e., non-mobile) software [12–14]. However, some of their findings may not directly apply to Android apps, or new findings might arise from the peculiarities of the Android platform.

To bridge this gap, we present an in-depth study of how developers create and use TDs when developing and testing Android apps. Specifically, the goal of our study is to get a better understanding of (1) how TDs are used in the Android ecosystem, and (2) whether TDs developed for Android apps differ from traditional TDs.

In our study, we first analyzed a dataset of 1,006 apps with publicly available test suites to collect information on the frameworks and approaches used to create TDs. This analysis shows that Mockito and Mockito-Kotlin are the most popular frameworks for creating TDs, with 33.5% of the apps in the dataset using either one

of these two frameworks. We then investigated several research questions by studying how TDs defined using these popular frameworks are created and used in the ten apps in the dataset that define the highest number of TDs using these frameworks.

Our results, based on the analysis of 2,365 TDs that replace a total of 784 classes, provide insight on the types of TDs used within Android apps and how they are utilized. In particular, they show that developers create TDs to replace both classes in the app and external classes, and that different kinds of TDs are indeed created, including stubs, mocks, and dummies. Our results also show that TDs used for testing apps differ in at least some respect from TDs used in traditional Java software. Specifically, our study found that there are different categories of TDs that are prevalent in this context, namely, TDs replacing classes in the Android framework, configuration classes, and GUI components. Whereas the first category is not surprising, the latter two provide evidence that, within Android apps, configurations are more common and classes are more tightly coupled with GUI elements than in traditional Java software. Finally, our results show that TDs can introduce test smells and even mistakes in test code, which motivates developing techniques to detect and eliminate these problems.

Contributions and Significance. To the best of our knowledge, this is the first study that classifies how developers use TDs, categorizing them based on their purpose and through both qualitative and quantitative analyses. We believe that our findings and their implications can inform future research in this area and help define automated or semi-automated techniques for better supporting developers in creating and maintaining TDs, ultimately improving the process of testing Android apps. Furthermore, our study infrastructure and experimental data are publicly available [15].

2 BACKGROUND

Android apps and their tests are mainly written in the Java or Kotlin programming languages [10]. These tests can run on either the JVM (JVM tests) or a device (*device tests*). Generally, JVM tests can include unit and integration tests, while device tests can include unit, integration, system, and GUI tests [10, 16]. Both JVM and device tests can use TDs to facilitate testing activities[1]. We define a TD as a developer-defined object that provides a (possibly partial) replacement for a class in the app or in an external library during testing. Within Android apps, TDs can replace classes defined in the app, classes from the Java library, classes defined in third-party libraries, and classes from the Android framework. Based on the functionality that the TDs provide to the test code, they can be classified [17, 18] into five main types: (i) *dummies*, (ii) *stubs*, (iii) *mocks*, (iv) *spies*, and (v) *fakes*. App developers can create TDs using test mocking[2] frameworks or by extending/implementing classes/interfaces. Among the frameworks that allow for creating TDs, there are both generic (e.g., Mockito [19], Mockito-Kotlin [20], PowerMock [21]) and specialized (e.g., OkHttp [22], Retrofit [23], Android Test Mock [24]) test mocking frameworks. The former allow for replacing classes of varying functionality, while the latter target classes offering a specific functionality (e.g., classes that connect to a web server). We now describe how developers can use generic test mocking frameworks to create TDs and then summarize the characteristics of the different types of TDs.

2.1 Generic Test Mocking Frameworks

When developers create a TD using a generic test mocking framework, they must first specify the class being replaced by the TD. To this end, developers can use initialization methods or annotations provided by the framework API (e.g., the `mock` method from the Mockito API [25]). After this step, developers can define stubbed method implementations for the TD and specify which method calls made to the TD should be verified during test execution. To stub a method, developers must specify (i) the method that should be stubbed, (ii) the arguments to which the stubbed method should respond, and (iii) the value/exception returned by the stubbed method. Generic test mocking frameworks offer API methods that can be combined to implement this functionality. For example, developers can use `when(td.m(arg)).thenReturn(val)` (based on the Mockito API) to specify that the TD `td` should return `val` when the method `m` is called with argument `arg` on the object. Developers can also use the framework API to specify the method calls that should be verified. For example, developers can use `verify(td).m(arg)` (based on the Mockito API) to check that (1) method `m` was called during test execution on the object `td` and (2) the argument passed to the method was `arg`. Finally, developers can use the APIs of these frameworks to create different types of TDs. In our work, the type of a TD is not identified by the API method used to create it (e.g., `mock` in Mockito), but rather by the functionality it provides.

2.2 Test Doubles Types

This section reports the definitions we use to characterize the different types of TDs, as formulated in related work [17, 18]. Due to space limitations, we do not provide here code examples for the different types but make them available in our online appendix [15].

Dummy: A dummy is an object that a test uses to exercise the component under test (CUT) but such that neither the test code nor the CUT access the object's state during test execution. Tests tend to use dummies to provide method parameters that are irrelevant for a specific test.

Stub: A stub is an object providing hard-coded (i.e., stubbed) answers when its callers invoke the object's methods during test execution. A stub might provide hard-coded answers only for some of its methods, and the answers are often specific to the intent of the particular test using the stub.

Mock: A mock is an object that offers a replacement for a class and such that some of the interactions with the object are verified during test execution. The verification task is defined in the test code but carried out within the mock object. A mock object might also provide hard-coded answers for some of its methods.

Spy: A spy is similar to a mock object in that some of the interactions with the object are verified during test execution. However, differently from a mock object, the primary operations for verifying the behavior of the spy are defined in the test code, rather than

[1] In this work, we discuss TDs of JVM and device tests, as this grouping is readily available through the source code of Android apps—generally, JVM tests are in the `test` folder and device tests are in the `androidTest` folder. We leave as future work the analysis of TDs in relation to how JVM and device tests can be divided into unit, integration, system, and GUI tests.

[2] Although these frameworks are informally called *mocking* frameworks, developers actually use the frameworks to create different types of TDs.

within the TD. Usually, these operations are encoded as developer-defined `assert` statements. As in the case of a mock object, a spy might also provide hard-coded answers for some of its methods. It is worth noting that this definition is consistent with related work [17, 18] but differs from the use of the term within Mockito [19]. Specifically, in Mockito, a spy is an object for which real method implementations are invoked during test execution unless they are stubbed. Because the two definitions focus on different aspects, our findings should not be directly mapped to Mockito's spies and should be instead interpreted based on the functionality provided by the TD.

Fake: A fake is an object that provides a working implementation for some of its methods but such that the implementation is made more efficient through "shortcuts" not suitable for production.

3 METHODOLOGY

To shed light on how Android developers create and use TDs, we investigated the following research questions (RQs):

- **RQ1: Which frameworks and approaches are most commonly used to create TDs?** This RQ aims to identify the most commonly used frameworks and approaches for creating TDs in the domain of Android apps. We use the findings from this RQ to scope the analyses of the remaining RQs.
- **RQ2: What types of classes do developers replace with TDs?** The goal of this RQ is to categorize the types of classes that are commonly replaced by TDs. In the RQ, we also provide a detailed analysis of the Android framework classes that are replaced by TDs.
- **RQ3: What TD types do developers create?** This RQ investigates the types of TDs used in the context of Android app testing. This RQ also analyzes whether developers use different types of TDs for different types of classes.
- **RQ4: How do tests use TDs?** While RQ2 and RQ3 characterize TDs through a manual inspection of the code associated with TDs, this RQ aims to characterize the runtime properties of TDs. Specifically, it investigates how tests use stubbed methods and how often interactions with TDs are verified.
- **RQ5: What problems can TDs introduce?** Because TDs are usually manually-created, they may introduce test smells or even errors in the test code. This RQ investigates issues emerging from the use of TDs.

The overall goal of these RQs is to inform researchers and practitioners and provide insights that can guide them in developing techniques and tools for creating, using, and maintaining TDs. To answer our RQs, we divided our study into two parts. First, we identified which frameworks and approaches developers most commonly use to create TDs (RQ1). Then, we studied how the TDs defined with the most popular frameworks and approaches are created and used (RQ2, RQ3, RQ4 and RQ5). The rest of this section describes the qualitative and quantitative analyses we performed to answer the RQs.

3.1 Frameworks and Approaches for TDs

In this section, we describe our methodology for answering RQ1. Specifically, we describe the dataset we used, the frameworks and approaches we considered, and the analysis we performed.

3.1.1 Dataset. To answer RQ1, we needed a dataset containing Android apps with a publicly available test suite. To the best of our knowledge, the dataset released by Lin and colleagues [10] is the most recent one satisfying this requirement, as it contains 1,002 apps with tests. These apps were mined from GitHub, and each app is available on at least one of 16 app markets (including the Google Play store [26]). When we cloned the app repositories, 972 of the 1,002 apps were still available on GitHub. To ensure our dataset does not include possibly trivial apps, we further filtered the dataset to only contain apps available on the Google Play store. After this step, the dataset contained 886 apps.

We performed a sanity check to verify that the apps have tests in their `test` and `androidTest` directories, which are the default locations used to store JVM and device tests [16], respectively. For this purpose we built an automated analysis on top of JavaParser [27] and ktlint [28] to traverse the abstract syntax tree (AST) of the test files looking for methods annotated with `@Test`, `@SmallTest`, `@MediumTest`, `@LargeTest`, or `@UiThreadTest`; we classified a test as any method having any of these annotations. Note that, by operating at the AST level, the analysis avoids considering tests in commented code. The analysis also excludes tests automatically created by Android Studio, which can be identified based on the name convention used by the IDE. Our analysis identified some apps without any meaningful test. Manual inspection confirmed that, at the time we retrieved them, those apps had no tests at all, had tests that had been commented out, or only had tests automatically created by Android Studio. After removing these apps, 833 apps remained in the dataset.

After manually inspecting the list of remaining apps, we observed that certain widely used apps, such as AnkiDroid [29, 30] (over five million downloads), were not present in the dataset despite being available in the curated list of open-source apps provided by F-Droid [31], which was considered in [10]. We noticed that these apps have multiple `AndroidManifest.xml` files [32], and that apps with these characteristic were excluded by Lin and colleagues [10]. Therefore, to avoid missing relevant apps, we decided to add apps (i) listed on F-Droid, (ii) available on GitHub, (iii) present on the Google Play store, and (iv) having meaningful tests. This resulted in the addition of 173 apps to the dataset, for a total of 1,006 apps. We used this dataset to answer RQ1.

3.1.2 Frameworks and Approaches Considered. In RQ1, we investigated how often developers create TDs (1) using either generic or specialized test mocking frameworks, or (2) extending/implementing classes/interfaces. To ensure we considered a comprehensive set of relevant frameworks, we performed a Google search using "*android test mocking*"[3] as the search terms and analyzed the first 100 results. Our online appendix [15] contains the complete search results. Based on the search results, we considered the generic test mocking frameworks EasyMock [33], jMock [34], Mockito [19], Mockito-Kotlin [20], MockK [35], and PowerMock [21], which all allow for creating TDs as described in Section 2.

We also considered Android Test Mock [24], MockServer [36], OkHttp [22], Retrofit [23], Robolectric [37], and RxAndroidBle [38] as additional, specialized frameworks. Android Test Mock provides

[3]We used the word "*mocking*" because developers and the documentation of multiple frameworks use this term to refer to test doubles in general.

stubs and mocks for ten specific classes of the Android framework. MockServer, OkHttp, and Retrofit support the creation of TDs for classes communicating with a web server. Robolectric allows for running tests interacting with the Android framework on the JVM by using a large set of classes that offer a simplified implementation of Android framework classes. Robolectric also allows app developers to implement their own replacements for Android framework classes; these developer-defined replacement classes are those we consider in this study. Finally, RxAndroidBle is a library that facilitates Bluetooth communications and offers support for replacing the framework's classes during testing.

3.1.3 RQ1: Which frameworks and approaches are most commonly used to create TDs? — Analysis. To identify the general and specialized test mocking frameworks used by a certain app, we first identified all the relevant *import* statements for each framework (e.g., `org.mockito` for Mockito), and then checked the import statements in the ASTs of the app's test files; if a test file used an import statement of a certain framework, we considered the app as using that framework. In that case, the analysis also computed the number of test files using that framework, so as to provide an indicative measure of the extent to which the framework was used by the project. It is worth noting that this measure could be computed differently, and possibly in a more accurate way (e.g., by considering all the API methods in the frameworks and identifying calls to these methods in the test code). However, we believe that this approximation is sufficient, as (1) we use this information only as a secondary measure, with the primary one being the number of apps using the framework, and (2) this measure does not affect the main findings of the study. To determine whether an app extends/implements classes/interfaces for creating TDs, we analyzed the ASTs of the app's test files and looked for classes that (i) contain "Dummy", "Stub", "Mock", "Spy", or "Fake" in their name, (ii) have a name that does not end with "Test" or "Tests", and (iii) are part of a file that does not use the import statements from the general and specialized frameworks we considered. This strategy is in line with an approach previously used in related work [13]. If an app had such a class, we considered the app as extending/implementing classes/interfaces for creating TDs.

3.2 Detailed Analysis of TDs

After investigating which frameworks and approaches developers use to create TDs, we identified Mockito and Mockito-Kotlin as the most popular frameworks for creating TDs in Java and Kotlin code (see details in Section 4). Consequently, we focused the remaining part of our study on these two frameworks. This part includes both manual, qualitative analyses and automated, quantitative analyses. We now describe our methodology to select the ten apps and detail the analyses we performed to answer the remaining RQs.

3.2.1 Apps. Our qualitative analysis focused on the ten apps with the highest number of TDs created using Mockito or Mockito-Kotlin and whose tests are maintained. We focused on ten apps due to the significant amount of manual effort involved in this part of the study, for both preparing the apps and performing the analysis. For example, even simply building the apps can be extremely time-consuming [39–41]. As for the analysis, there are many tasks that involve a significant manual work, including classifying the types of classes replaced by TDs and manually identifying the types of TDs. Although focusing on a smaller set of apps may hinder the generalizability of our results, as we also discuss in Section 6, it allowed us to perform a detailed analysis of how developers create and use TDs and get valuable insights.

To identify the number of TDs in an app, we (1) analyzed the Mockito and Mockito-Kotlin APIs [20, 25], (2) identified API methods (e.g., `mock`) and annotations (e.g.,`@Mock`) that can be used to create TDs, (3) parsed the ASTs of the test files in the app to collect the locations using such methods or annotations, and (4) counted the number of such locations. To identify whether an app's tests were maintained, we analyzed the app's repository, counted the number of commits of the test files in the year preceding the beginning of our study (August 2020), and considered the tests to be maintained if the app had one commit per month on average on the test files. The rationale for using this second criterion is that tests that are maintained are more likely to be relevant. Table 1 reports the ten apps we selected based on this strategy. For each app, the table provides an identifier (ID_A), the app's name (*Name*), the app's category as listed on the Google Play store (*Category*), the app's version considered (*Version*), the lines of code (in KLOC) for the app's source files (*SL(K)*), the lines of code (in KLOC) for the app's test files (*TL(K)*), and the number of TDs in the app created using Mockito or Mockito-Kotlin (*Total* under the *Test Doubles* header). It is worth noting that six of the ten apps also use additional test mocking frameworks beside Mockito or Mockito-Kotlin. Specifically, six apps (A02, A04, A06, A08, A09, and A10) create 12 class replacements using Robolectric, two apps (A08 and A09) create 30 TDs using Powermock, and one app (A06) creates one TD using OkHttp. Since our analysis is based on 2,365 Mockito/Mockito-Kotlin TDs, we believe that considering the few TDs created using other frameworks would impact the results only marginally.

3.2.2 RQ2: What types of classes do developers replace with TDs? — Analysis. To answer RQ2, we performed four analyses. First, we characterized the functionality provided by the classes. Second, we identified whether the classes belonged to the app, the Java library, third-party libraries, or the Android framework. Third, we studied the dependencies of those classes that are defined in the app. Finally, we performed a categorization of the classes that are replaced by TDs and are part of the Android framework.

The first one is a qualitative analysis that combines deductive, inductive, and axial coding [42, 43]. Deductive coding is a systematic approach for manually coding (i.e., labeling) textual content starting from an already available set of codes (i.e., labels). Inductive coding derives new codes based on a systematic analysis of the text data. Axial coding relates codes to one another and finds higher-level codes that represent abstractions of the original codes.

In our analysis, a code is a label that categorizes the functionality provided by a class, which we inferred by analyzing the source code and the documentation of the class. We also analyzed any class dependencies that may help clarify the class functionality and the code of the TD replacing the class. Specifically, we first looked at the test code using the TD to identify the part of the app being tested. We then focused on the class being replaced by the TD and inspected the name of the class, imported dependencies, declared

Table 1: Characteristics of the ten apps (and their tests) considered in the second part of our study.

ID_A	Name	Category	Version	SL (K)	TL (K)	Test Doubles			PC Analysis		TDT Analysis		Tests		
						Total	JVM	Device	Total	CB	Sample	CB	Total	JVM	Device
A01	andFHEM	Personalization	6.0.2	25.2	5.6	70	70	0	28	10	60	10	587	585	2
A02	AnkiDroid	Education	2.13	52.8	7.0	60	60	0	32	11	53	7	341	274	67
A03	AnySoftKeyboard	Tools	1.1	28.7	21.6	166	166	0	53	17	117	23	1,038	1,038	0
A04	Nextcloud	Productivity	3.12.1	65.1	6.5	116	108	8	57	19	90	16	1,142	1,032	111
A05	OpenSRP	Medical	1.0.14	19.7	2.5	153	153	0	85	29	110	22	56	56	0
A06	StreetComplete	Travel & Local	21.2	27.4	8.0	179	166	13	63	20	123	25	770	664	106
A07	Travel Weather	Travel & Local	1.5.1	3.2	1.9	104	104	0	55	19	83	15	128	128	0
A08	WiFi Analyzer	Tools	2.1.2	8.0	10.7	193	193	0	78	25	129	28	706	706	0
A09	Wikimedia Commons	Photography	2.13	24.3	4.3	222	222	0	81	26	141	31	270	246	24
A10	WordPress	Productivity	15.2.1	135.3	31.3	1102	1098	4	252	83	286	154	1,514	1,396	118
						2,365	2,340	25	784	259	1,192	331	6,553	6,125	428

methods, used variables, and provided code comments. We also used the same procedure to inspect the classes used by the class in the case that this operation was necessary to better understand the functionality provided by the class.

Overall, we analyzed 784 classes. Table 1 reports the number of classes analyzed for each app (column *Total* under the *PC Analysis* header). These are all the classes associated with the TDs we considered, which we identified by statically analyzing the compiled code of the tests. Specifically, we compile the tests, retrieve the locations where developers initialized TDs (e.g., where developers use the `mock` method), and extract the class types associated with the objects. We built this analysis on top of Soot [44].

Our qualitative analysis is divided into three parts and performed by two raters, which are two of the paper authors. In the first part of the analysis, the two raters analyzed a sample of 259 classes to define the analysis codebook—a document detailing, for each code, the set of rules specifies the characteristics that should be observed to assign a code to a class. The set of rules also includes typical examples of classes having a specific code. The sample size used to create the codebook was created using stratified random sampling and is statistically significant with a 95% confidence level (CL) and a 5% margin of error (ME). Table 1 reports the sample sizes we used to create the codebook (column *CB* under the *PC Analysis* header).

The two raters used the categories identified by related work [12, 13, 45] as the initial set of codes for the codebook (deductive coding). In the process of analyzing the classes in the sample, the raters increased the number of categories to 28 (inductive coding) and then grouped the categories into five main groups (axial coding). This iterative part of the analysis took the raters around two person-months to complete. Table 2 reports the codes produced by this part of the analysis. The entire codebook we used is available in our online appendix [15]. Our analysis produced two categories that are not present in related work: *configuration* and *GUI component*. We believe that these new categories emerged because the software domain we target is characterized by aspects (e.g., GUI components) that are not a key part of the software domains analyzed in related work [12, 13, 45]. Conversely, our codebook does not include some of the categories identified in related work—*java library* and *external dependencies*—because we distinguish between classes in the app, the Java library, third-party libraries, or the Android framework later in an orthogonal categorization. Finally, our codebook contains category *generic*, for classes whose functionality did not fall into a big enough category during the axial coding analysis. This category includes classes labeled as *domain objects* in related work [12, 13, 45] and can also include classes from the Android framework or external libraries that can be considered as domain objects when the framework or libraries are considered in isolation.

Table 2: Codes used to categorize the classes replaced by TDs.

Code	Summary Description
Configuration	Class used to manage the app's settings.
Database	Class that performs database operations.
GUI Component	Class that is part of the app's GUI.
Networking	Class that perform network operations.
Generic	Class that provides a functionality not falling in the other categories.

After creating the codebook, the two raters analyzed 10% of the remaining classes using the codebook (i.e., they used the codebook rules to categorize the classes), and we measured their inter-rater reliability (i.e., the degree of agreement among raters in the analysis) using the Krippendorff's alpha coefficient [46, 47]. Based on the codes assigned by the two raters, the alpha value was 0.88, which indicates high reliability. After discussing and resolving mismatching codes, the two raters proceeded with the last part of the analysis and coded the remaining classes. Given the high value of their inter-rater reliability, they equally split the remaining classes and coded them independently.

After finishing the coding process, the raters also identified whether each analyzed class belonged to the app, the Java library, a third-party library, or the Android framework. Then, for each of the classes in the app, we identified whether the class was directly coupled with the Android framework by checking its dependencies using an AST parser that analyzes the import statements in the class. Finally, we identified the most recurring classes from the Android framework replaced by TDs and performed a detailed categorization based on their containing package.

3.2.3 RQ3: What TD types do developers create? — Analysis. To characterize the types of TDs that appear in the test code, we conducted a qualitative analysis based on deductive coding, where the code indicates a type of TD. To assign a code to a TD, we studied the functionality of the TD by inspecting the test code, by focusing on the methods in the Mockito API and on assertion statements. For example, if the test code only creates the TD object without specifying any additional behavior for it, we would classify the TD as a dummy. As another example, if the test code creates the TD object and stubs one of its methods (e.g., using the `when(x.m()).thenReturn(y)` construct from Mockito), we would classify the TD as a stub. Because different tests might define a different behavior for the same TD

Table 3: Codes used to categorize the types of TDs.

Code	*Summary Description*
Dummy	The behavior of the test double is not stubbed nor verified.
Stub	The test double offers stubbed method implementations.
Mock	The interactions with the test double are verified.
Spy	The test verifies the test double's interactions using assertions.
Fake	The test double provides a simplified implementation.

object (e.g., when the TD is created as a test class attribute), our analysis might assign multiple codes to the same object.

We split this coding process into three parts, as we did for the qualitative analysis of RQ2, and the same two authors that performed the analysis of RQ2 performed this analysis as well. In the first part of the analysis, the raters analyzed a statistically significant sample (CL=95% and ME=5%) of 331 TDs to define the analysis codebook. Table 1 reports the sample sizes we used to create the codebook (column *CB* under the *TDT Analysis* header). Table 3 reports the codes used in our codebook and their summary descriptions. The entire codebook is available in our online appendix [15].

After creating the codebook, the two raters labeled a statistically significant (CL=95% and ME=5%) sample of TDs for each app, excluding samples already labeled when creating the codebook, for a total of 1,192 TDs. The sample sizes per app are reported in Table 1 (*Sample* column). We analyzed statistically significant samples instead of the whole dataset because the effort required to do so would be considerable (estimated at around four person-months). In the second part of the analysis, the raters coded 10% of the TDs in the samples, and we measured their inter-rater reliability. Based on the coding results, the Krippendorff's alpha value was 0.97, which indicates high reliability. As for RQ2, after discussing and resolving mismatching codes, the two raters split the remaining TDs and coded them independently.

After categorizing the types of TDs, we combined the results from RQ2 and this RQ to understand how the types of TDs relate to the type of class they replace.

3.2.4 RQ4: How do tests use TDs? — Analysis. To further characterize key properties of TDs, we analyzed how tests use TDs by running the tests of the apps with an instrumented version of Mockito[4] while collecting various data. As the tests ran, our instrumentation logged the calls made to the methods of the TDs, identified which calls were made to stubbed methods, and recorded how many of these calls were being verified during test execution. The instrumentation also computes various properties of these methods: how many unique methods were being stubbed, the location in which these methods were stubbed, and whether methods return hard-coded values or intended exceptions. Table 1, in the *Tests* section on the right, reports the number of executed tests, both overall (*Total*) and grouped by test type (*JVM* or *Device* tests).

3.2.5 RQ5: What problems can TDs introduce? — Analysis. To answer RQ5, our analysis identified *unnecessary stubs*—stubbed method never called during test execution—and *mismatched stubs*—stubbed methods called with arguments that differ from those specified for the stub (e.g., a stub `td` specified as `when(td.m(2)).thenReturn(3)` and then called by a test as `td.m(4)`). Although these issues might not lead to test failures, these problems often indicate potential issues in the underlining test code. Unnecessary stubs, in particular, may indicate superfluous, dead, or outdated code in the tests. Furthermore, both unnecessary and mismatched stubs may indicate tests that are not checking for the intended behavior of the CUT. To identify these kinds of stubs, we ran the tests with the stubbing hints option of Mockito enabled [48], by adding a test rule to the tests. It is worth noting that the next major release of Mockito will notify developers when these problems occur [49], which indicates that they are perceived as potentially relevant issues.

[4]Because Mockito-Kotlin internally relies on Mockito, our Mockito instrumentation worked for it transparently.

Table 4: Frameworks and approaches considered in our study together with their occurrences in our dataset of 1,006 apps.

Type	*Framework/Approach Name*	*Apps*	*Occurrences*
Generic Test Mocking Frameworks	EasyMock	2	3
	jMock	0	0
	Mockito	323	2123
	Mockito-Kotlin	55	605
	MockK	17	108
	PowerMock	41	148
Specialize Test Mocking Frameworks	Android Test Mock	18	24
	MockServer	0	0
	OkHttp	42	137
	Retrofit	4	5
	Robolectric	29	87
	RxAndroidBle	0	0
Extend/Implement Classes/Interfaces	-	68	146

4 RESULTS

In this section, we present the results of our study on how developers create and use TDs when testing Android apps.

4.1 RQ1: Which frameworks and approaches are most commonly used to create TDs?

Table 4 shows how many of the 1,006 apps in our dataset use the frameworks and approaches we considered. For each framework/approach, the table reports its name (*Framework/Approach Name*), the number of apps with tests that use the framework/approach (*Apps*), and the number of files using the framework/approach (*Occurrences*). For Robolectric and the approach based on extending/implementing classes/interfaces, the number of occurrences identifies the number of developer-defined TD classes. Of the 1,006 apps considered, 397 apps (39%) use either a framework or an alternative approach to create TDs. (Adding the number of apps in Column *Apps* results in a higher number because some apps use more than one approach to create TDs, and thus appear in more than one row.)

Mockito is the most used framework, with 323 apps and 2,123 test files using it. This result is in line with the findings from related work [50], which identified Mockito as the most popular framework for Java-based projects. Our results also highlight that Mockito-Kotlin finds a significant adoption in Android apps, with 55 apps and 605 test files using that framework. We believe that this result is due to the fact that Kotlin is gaining popularity among the languages used to develop Android apps [51–53]. The total number of apps using either Mockito or Mockito-Kotlin is 337, which accounts for 33.5% of the apps in our dataset. After further analyzing the test code of these apps we found that developers use the two frameworks

more frequently in JVM tests than in device tests. Specifically, we observed that 39.1% of the 758 apps with JVM tests use one of the two frameworks within these tests. This is in contrast with what happens for the 562 apps with device tests, where only the 9.1% of the apps have tests that use either one of the two frameworks. Furthermore, among the apps using Mockito or Mockito-Kotlin, there are 7,303 TDs defined across 18,747 JVM tests, and 315 TDs defined across 3,524 device tests. This big gap seems to indicate that TDs are a relevant aspect of JVM-based testing of Android apps, whereas they play a smaller role in the context of device tests.

Our results also show that generic test mocking frameworks find a wider adoption than specialized test mocking frameworks. Specifically, 35.9% of the apps use a generic test mocking framework, while only 6.2% of them use a specialized one. Note that, although developers do not often use Robolectric to create manually-defined TDs (only 2.8% of the apps defines such TDs), they use the TDs already provided by the framework more extensively; 17.3% of the apps in our dataset have tests that rely on those TDs. When we analyzed apps with tests that create TDs by extending/implementing classes/interfaces, we found that only 6.8% of the apps in our dataset use this approach. Our analysis also revealed that 67.6% of them also use a generic test mocking framework. This result seems to suggest that, at least in some cases, developers find it necessary to create ad-hoc TDs in addition to those they create using test mocking frameworks, which may indicate the need for additional features within these frameworks.

Finally, by comparing apps that use a framework or some alternative approach to create TDs and apps that do not, we observed that the average number of tests for the apps in the former category is 65.1 and the median is 22, while for the latter category the average is 13.9 and the median is 5. A Mann-Whitney U test (95% CL) shows the difference between the two groups to be significant. As we further discuss in Section 5, we believe this is a potentially interesting result that deserves further investigation.

RQ1 answer: Mockito and Mockito-Kotlin are the most widely used frameworks, with 33.5% of the apps using either one of the two frameworks. Furthermore, generic frameworks find a wider adoption than specialized frameworks or approaches. Finally, some apps use multiple approaches, which may indicate the need for extending the individual approaches.

4.2 RQ2: What types of classes do developers replace with TDs?

Figure 1, Figure 2, and Table 5 present the main results of the analyses we performed to answer RQ2. Fig. 1 reports the categorization for the types of classes replaced by TDs, showing the percentage of each category for each app and the number of classes in each category. For the apps we considered, the generic category includes the highest number of classes replaced by a TD. This result is in line with related work [12, 13, 45] (as we included domain objects in this category) and is expected, as this category is the broader category among those we considered.

The remaining categories account for 32.5% of the classes we analyzed, with the GUI component category being the most frequent and including 10.7% of the classes. All the classes in this category were replaced by TDs in JVM tests. The remaining three categories (database, networking, and configuration) include classes that provide access to external resources. All the apps creating TDs for either one of these categories do so for multiple classes (e.g., WordPress (A10) creates TDs for 31 classes accessing the network).

After analyzing the types of classes replaced by TDs, we investigated whether those classes are defined in the app, the Java library, third-party libraries, or the Android framework. Figure 2 illustrates the results of this analysis. Across all apps, there is an approximately equal balance between the classes defined in the source code of the apps and those defined in either third-party libraries or the Android framework. Specifically, 54.6% of the classes that are replaced by TDs are defined in the apps' source code, and 43.5% of them are defined in external dependencies. Our analysis also revealed that 90% of the classes defined in the app's source code and replaced by TDs have external dependencies, and for 63.1% of those, the dependencies involve the Android framework. This result differs from related work analyzing mocking in traditional Java programs [12, 45], where the percentage of classes replaced by TDs and with external dependencies is lower than 60%[5]

Furthermore, 19.4% of the classes replaced by TDs belong to the Android framework. Table 5 reports, for the ten most recurring packages that contain those classes, the number of unique classes from the packages (column *Classes (#)*) and the number of times that those were replaced by a TD (column *Occur. (#)*). The package containing the highest number of classes and occurrences is `android.content`, which contains classes used to share content between application components through the framework. For example, classes `android.content.Context` and `android.content.Intent` were replaced by TDs to allow test code to retrieve specific application data during test execution. The top packages also include `android.location`, which provides classes for location-based services. The classes from this package that were replaced by TDs provide specific location information or facilitate access to the information during testing.

Among the Android framework classes replaced by TDs, none are from the `android.hardware` package, which contains camera and sensor classes, even if three apps (A02, A06, and A10) use classes from this package. We find this result interesting and believe that suitably replacing those classes might help in producing better test suites. We additionally observed that the six apps that use Robolectric (A02, A04, A06, A08, A09, and A10) also replace classes defined in the Android framework, suggesting that better Robolectric models may be needed because either they do not include some commonly used classes or, if they do, they are not used.

RQ2 answer: Developers replace classes that fulfill domain logic (67.5%), model GUI components (10.7%), access the network (8.7%), perform database operations (8.5%), and provide app configurations (4.6%). In a large number of cases (90%) developers create TDs for classes that are external or coupled with external dependencies. Developers also replace Android classes to be able to access specific app data during testing.

[5]We computed this number by aggregating the results from RQ1 in [12].

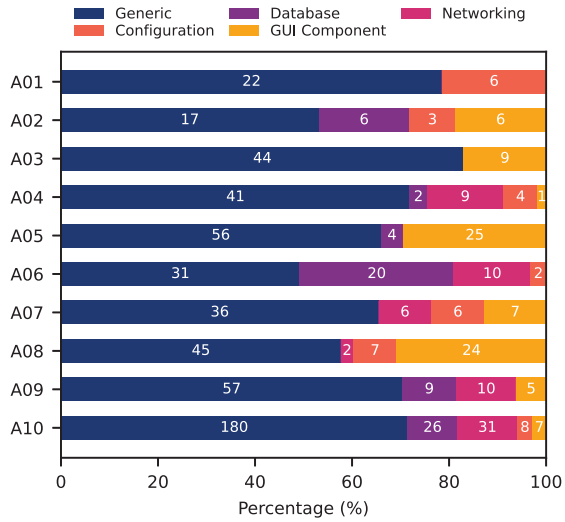

Figure 1: Types of classes replaced by TDs.

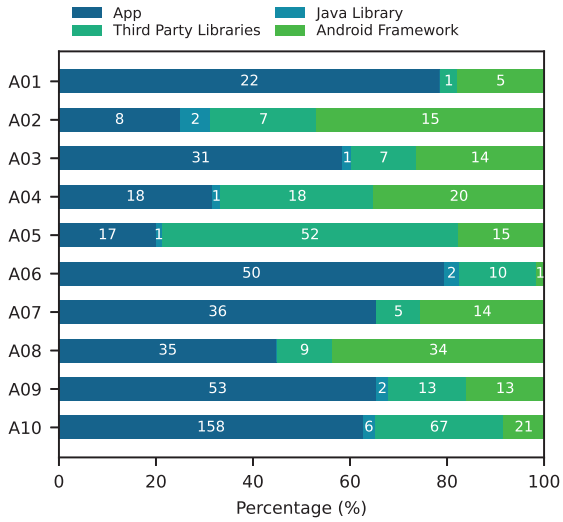

Figure 2: Location of the classes replaced by TDs.

Table 5: Packages of classes from the Android framework frequently replaced by TDs.

IDp	*Package*	*Classes (#)*	*Occur. (#)*
P01	android.content	10	36
P02	android.widget	10	10
P03	android.view	7	15
P04	android.app	6	12
P05	androidx.lifecycle	6	9
P06	android.os	5	7
P07	android.location	4	5
P08	androidx.fragment.app	4	4
P09	android.net	3	9
P10	android.content.res	3	7

4.3 RQ3: What TD types do developers create?

Figure 3 shows the different types of TDs in the apps we considered. Figure 3 displays the relative frequency for each type and is based on the statistically significant samples of Table 1 (*TDT analysis* section). Across all apps, we identified 28 unused TDs (e.g., attributes annotated with `@Mock` but never used by any test), which are not reported in Figure 3. For this reason, and because we might assign more than one type to a single TD (e.g., when a TD is used differently by different tests), the total number of types in the figure might differ from the sample size reported in Table 1. Our results show that the apps do use different types of TDs, and that dummies, stubs, and mocks are the most prevalent types of TDs. Specifically, 39.9% of the TDs are dummies, 32.9% are stubs, 26% are mocks, and only 1.2% are spies. Notably, our analysis did not identify any fakes in the ten apps we considered. This result was expected, as these apps rely on Mockito and Mockito-Kotlin, which do not support the creation of fakes. Overall, these numbers show that developers often define stubbed implementations for methods but also verify interactions between components under test and the TDs.

To provide a different view on these data, the first part of Table 6 reports the number of dummies, stubs, mocks, and spies for each type of class identified in RQ2. A chi-squared test at a significance level of 5% rejected the null hypothesis that TDs types are independent from the types of classes. The second part of Table 6 presents TD types with respect to the classes grouped based on where they are defined. Also in this case, a chi-squared test at a significance level of 5% rejected the null hypothesis that TDs types are independent from classes grouped by location.

Among the two categories that are not present in related work, GUI component and configuration, the most frequent types of TDs are mocks (40%) and stubs (38%), respectively. Mocks for the GUI components are mostly used to verify interactions that should or should not happen, whereas stubs for the configuration components allow the tests to retrieve specific configuration values.

RQ3 answer: Dummies (39.9%), stubs (32.9%) and mocks (26%) frequently occur in the tests of the Android apps we considered. This seems to indicate that, although a large number of TDs are trivial classes created simply to allow the tests to run, developers also (1) make extensive use of stubbed implementations and (2) frequently use TDs to verify interactions.

4.4 RQ4: How do tests use TDs?

Table 7 reports the results of the dynamic analysis described in Section 3.2.4. The table shows the characteristics of the calls made by the tests on both stubbed and verified methods. For each app, it reports the following information: number of TDs whose methods are called at least once by the tests (*TD*, for both stubbed and verified methods), total number of calls to stubbed methods made by the tests (*SMC*), total number of locations in the tests that make calls to stubbed methods (*CL*), number of unique methods that are stubbed at least once, whether they are called or not by the tests (*SM*), number of different locations in which any method is stubbed (*SL*), total number of stubbed methods returning values (*VR*), total number of stubbed methods returning exceptions (*ER*), and total

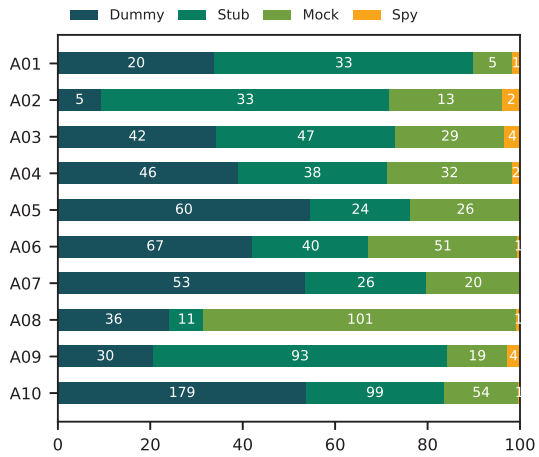

Figure 3: Types of TDs (percentages).

number of method calls verified (*VMC*). In total, the tests made 27,111 calls to stubbed methods. Analyzing the 1,493 code locations making these calls and the 782 unique methods being called, we noticed that tests tend to rely heavily on a small subset of the stubbed methods. For example, for three of the apps (A01, A03, and A04), a single stub accounts for over 50% of the calls to stubbed methods. We also inspected the code defining and using the three most called stubbed methods for each app and observed that the majority of these methods are stubbed to improve test execution performance (e.g., to avoid reading from configuration files).

Table 7 also shows that developers stubbed the same methods at different code locations. Specifically, the 782 unique stubbed methods are stubbed in 2,174 different locations, and 423 of these methods are stubbed more than once, which seems to indicate that tests use stubbed methods for different purposes. Furthermore, although the majority of stubbed methods return values (column *VR*), some of them return exceptions (column *ER*).

The numbers in the table for the verified methods show that 2,008 TDs were used to verify 3,492 method calls. Considering that the total number of test execution is 6,553 (see Table 1), this roughly correspond to one verified method call for every other test execution (on average). This result further confirms our findings from RQ3 that TDs are frequently used to verify interactions.

RQ4 answer: Tests perform a large number of calls to stubbed methods (27,111 calls across 6,553 test executions). Many of these calls involve stubs created to improve performance and speed up test execution. Methods are often stubbed at multiple locations, indicating that tests may stub the same method differently for different purposes.

4.5 RQ5: What problems can TDs introduce?

The analysis we discussed in Section 3.2.5 revealed that all the apps we considered contain unnecessary stubs. In many cases, this happens because the test code that creates stubs is overly general, and stubs are created also by tests that do not actually need them. The most extreme example of this issue is in app AndFHEM (A01),

Table 6: TD types for the different class types (abs. values).

Category	*Dummy*	*Stub*	*Mock*	*Spy*
Configuration	33	38	29	0
Database	47	63	40	0
GUI Component	55	37	64	4
Networking	65	56	40	1
Generic	501	369	257	15
App	493	369	288	9
Java	11	4	5	1
Third Party Lib.	26	69	16	1
Android	171	121	121	9

Table 7: Characteristics of the calls made by the tests on both stubbed and verified TD methods.

ID_A	*Stubbed*							*Verified*	
	TD	*SMC*	*CL*	*SM*	*SL*	*VR*	*ER*	*TD*	*VMC*
A01	368	1557	47	35	84	84	0	5	5
A02	130	395	46	35	60	54	6	36	133
A03	596	11104	97	41	128	121	7	626	1445
A04	208	3779	55	37	76	74	2	139	236
A05	95	599	292	44	86	86	0	46	48
A06	405	711	79	64	210	181	29	180	206
A07	113	185	55	38	69	61	8	35	44
A08	406	659	230	144	404	393	11	497	781
A09	115	135	43	43	123	116	7	70	86
A10	1740	7987	549	301	934	932	2	374	508
	4,176	27,111	1,493	782	2,174	2,102	72	2,008	3,492

in which a method in the test code defines a stub for each of the 767 resource strings [54] of the app and is called by 135 tests that do not actually need the stubs. Overall, in the tests we considered, 106,545 unnecessary stubs are created at 624 test code locations. It is worth noting that this problem can be seen as an instance of the general fixture test smell [55].

Although not as prevalent as unnecessary stubs, our analysis also revealed 19 issues related to mismatched stubs in 4 different apps (A02, A07, A08, and A10). Mismatched stubs are problematic because a test may exercise a behavior different from the intended one and still pass. For example, in AnkiDroid (A02), a test meant to exercise specific lines in the code never reaches them because the call to a mismatched stub returns a value different from the expected one, which causes the execution of a different control flow without affecting the outcome of the test. We provide a full discussion of this issue in our online appendix [15].

RQ5 answer: The 106,545 unnecessary and 19 mismatched stubs reported by our analysis provide evidence that TDs can lead to test smells and to the testing of functionality that differs from the intended one.

5 DISCUSSION AND ACTIONABLE INSIGHTS

In this section, we summarize the main findings of our study and discuss some insight and actionable items derived from them.

Importance of TDs in Android testing. Before this study, it was not known how frequently Android apps use a framework or some alternative approach to create TDs. Our study shows that

a considerable percentage (roughly 40%) of the apps that contain automated tests use at least one of those frameworks or approaches. *This result motivates the investigation and development of techniques that support developers in creating and maintaining TDs.* The study also finds that these apps tend to have a larger test suite as compared to apps that do not use TDs. We believe that it is worthwhile to perform additional studies, possibly including interviews to developers, to assess whether this is just a correlation or it instead indicates that extensive testing of an app is likely to require the use of TDs. The latter would provide an even stronger motivation for the development of techniques that support the creation and maintenance of TDs.

Supporting Mockito and Mockito-Kotlin. Similar to what was found by studies on Java projects [50], we identify that Mockito and Mockito-Kotlin, which are used by one third of the apps in our dataset, are the most frequently used frameworks. However, comparing the adoption of these technologies between Android and Java projects, our results show that these two frameworks are more widely used within Android apps than Mockito within traditional Java projects (23% adoption rate) [50]. One possible explanation for this difference is that *Android apps are more tightly coupled with their external dependencies [5, 56–58], and it is therefore necessary to account for these dependencies during testing.* In fact, among the ten apps we considered in our detailed analysis, a majority (90%) of classes replaced by TDs either are defined in external dependencies or use external dependencies. This is in contrast with what was identified by related work [12, 45] in the domain of Java programs, where this percentage was 60%. These results, in addition to motivating the development of techniques for creating and maintaining TDs, *also indicate that the techniques would be mostly useful if they would support Mockito and Mockito-Kotlin.*

Helping developers create TDs. When creating tests, developers must decide which parts of a system to replace with TDs and which TDs to use [12, 14, 45]. We believe that the results of our study, and possibly further studies along similar lines, can help guide the development of recommender systems that help developers identify classes that should be replaced by TDs.

As a first observation, our results show that Android developers use different types of TDs, and that stubbed implementations and mocks that verify interactions between code under test and TDs are prevalent. As far as stubs are concerned, we observe recurring patterns. In particular, developers stub methods for data communications that are hard to setup (e.g., communications with classes in the `android.content` package) or for specific types of data (e.g., data associated with classes in the `android.location` package). Developers also create stubs to improve test execution performance. As for TDs that verify interactions between TDs and components under test, we find that this is done for all the types of classes we analyzed and that both interactions that should and should not happen are verified. Based on the results, we believe that *techniques that support creating and maintaining TDs should focus on stubs—helping developers identify which methods require stubbing and what values should be returned by these stubs—but also on mocks—helping developers also decide which interactions to verify.*

Identification of which methods to stub could be done by analyzing how the data is generated within the method (e.g., whether it is location dependent) or by examining the performance cost of different methods called during testing. This latter case is particularly important to ensure that JVM tests run quickly, as that is one of the goals of those tests [59]. As for the values (or exceptions) that should be returned by the created stubs, test carving techniques [60–62] could be used to identify, record, and suggest values flowing between the boundaries of tests and code under test. Similarly, approaches that analyze the interactions between tests and code under test could be used to identify interactions that should and should not happen, create corresponding checks, and suggest them to developers.

Android-specific TDs. An additional way in which our results could be leveraged to develop techniques that support Android developers is by analyzing the Android-specific TDs that are used in the apps. Specifically, our analysis of the different types of classes that are replaced by TDs identifies two categories of classes that are characteristic of Android apps: configuration and GUI component. *Because configuration and GUI component classes are typically part of the Android framework or inherit from classes therein, they can be easily identified and proposed to the developer as possible candidates for replacement by TDs.* Furthermore, our study found that a large percentage of classes replaced by TDs consists of classes that either are external dependencies or use external dependencies, and that this happens more frequently than for traditional Java programs [12, 45]. One possible explanation is that Android apps tend to have a tighter coupling with their external dependencies [5, 56–58]. *Additional studies focused on the coupling information between apps and their external dependencies may help identify which classes should be replaced by TDs.*

TDs in JVM and device tests. Android developers can use TDs in both JVM and device tests [10, 16]. Our study identifies a noticeably larger number of TDs—in particular, Mockito and Mockito-Kotlin TDs—in JVM tests as compared to device tests. Although this is not surprising, as JVM tests are run without a complete Android framework and might therefore need to account for the missing elements (even when Robolectric is used as the library provides a partial implementation of the Android framework [37, 63]), it is interesting to observe such a large difference. Based on these findings, *we recommend prioritizing the design of automated techniques for supporting the creation and integration of TDs in JVM tests*, as those are likely to find larger adoption in practice.

Furthermore, future work could investigate the reasons behind these differences. Analyzing the 25 TDs in the device tests for the 10 apps in Table 1, in particular, we found that all of them occur within integration tests, none is used within GUI tests, and 17 of them are checking for interactions happening with the TDs. This was less expected because, for instance, GUI tests would typically interact with external services (e.g., a backend server or a database) and would therefore benefit from the use of TDs. Based on these preliminary data, we hypothesize that developers may prefer to avoid TDs in device tests, in order to have higher-fidelity tests, and only use them for specific purposes (e.g., verifying that some calls happen during testing, rather than replacing components in the system). Interviews with app developers may help confirm or refuse these hypotheses and, more generally, shed light on why TDs are less used in device tests.

Supporting debugging of TDs. Like all activities that involve a considerable amount of manual effort, creating and maintaining

TDs in Android apps is error prone. In fact, our study identifies cases of faulty TDs and instances of test code smells related to the usage of TDs (see Section 4.5), which motivates the development of techniques that support developers in debugging TDs. Based on our results, *a starting point could be the development of techniques that identify obsolete TDs, which could be done by identifying TDs that are not actually exercised during testing and by analyzing code and tests co-evolution.* It is worth noting that, although our study highlights these issues in TDs for Android apps, they might also appear in other types of software, so other application domains could also benefit from these techniques.

6 THREATS TO VALIDITY

As it is the case for most empirical studies, there are threats to validity associated with the results we presented. In terms of external validity, our results might not generalize to other Android apps and corresponding tests. In RQ1, we mitigated this threat by considering the largest (to the best of our knowledge) dataset of apps with publicly available test suites in the literature, with apps that vary widely in terms of size and category. For RQ2–RQ5, we chose to perform our in-depth analysis based on the ten apps with the highest number of TDs due to the significant manual effort involved in preparing the apps for the analysis and performing the analysis, as we discussed in Section 3.2.1. Although this allowed us to perform a detailed investigation on over 2,000 TDs and carefully inspect the results and the corresponding code, we acknowledge that this part of the analysis is a case study. Additional studies based on more apps, possibly selected using a different sampling strategy, are needed to confirm the validity of our results and gather further insights into how developers create and use TDs.

In terms of construct validity, our results might be affected by errors in the implementation of the tools we used to perform our analyses. To mitigate this threat, we extensively tested our tools and manually inspected our results. Finally, we also performed qualitative analyses, which might be characterized by divergent understanding among the raters. We are confident in the reliability of our analysis as the inter-rater reliability we measured was considerably high.

7 RELATED WORK

Other researchers performed empirical studies on Android app testing [6–11]. Specifically, some work observed that developers use testing frameworks such as JUnit, Robolectric, and Robotium [7]. Other work confirmed that most apps are still poorly tested, although test automation and test quality are improving along with the increasing success and wide adoption of mobile apps [8–10]. Yet other work showed that many apps had at least one flakiness issue in their lifetime, and that the environment is one of the main causes of flakiness together with concurrency [11]. None of this body of work focuses on how Android developers use TDs within their test suites.

Other researchers, however, have studied test mocking practices in non-mobile projects [12–14, 45, 50, 64]. Spadini and colleagues [12] analyze over 2,000 mocks objects in 4 Java projects and report that the usage of mocks highly depends on the responsibility of the class, and that developers frequently mock dependencies that make testing difficult. Their study also shows that mocks tend to exist since the very first version of the test class and tend to stay for its whole lifetime. Pereira and Hora further explore this topic by analyzing 12 popular Java software projects, distinguishing mock objects from mock classes, and further classifying which classes developers mock [13]. Similarly, Zhu and colleagues study over 10,000 tests in 4 open-source projects and propose a tool, MockSniffer, for identifying and recommending mocks for unit tests [14]. Additionally, the work from Trautsch and colleagues[64] focuses on mocking practices in 10 Python projects. To the best of our knowledge, none of the studies on mocking practices (1) differentiates uses of TDs as we do in this paper, (2) focuses on mobile apps, or (3) aims to identify possible issues with TDs. Our results show, for instance, that Android apps replace types of classes that were not categorized before and highlight that both stubbing and operations to verify method calls are frequent and important. Our study also shows the need for better techniques for debugging and maintaining TDs.

Finally, related work also focused on generating, using, or maintaining test mocks automatically [1–3, 5, 65–73]. Our paper provides specific insights for researchers who want to define approaches along these lines for in the context of Android.

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grants CCF-1563991 and CCF-0725202, Spanish Government's SCUM grant RTI2018-102043-B-I00, the Madrid Regional project BLOQUES, DARPA, under contract N66001-21-C-4024, ONR, under contract N00014-18-1-2662, DOE, under contract DE-FOA-0002460, and gifts from Facebook, Google, IBM Research, and Microsoft Research.

8 CONCLUSION

In this paper, we presented an in-depth study aimed to understand how developers create and use TDs in Android apps. Our results showed that Mockito and Mockito-Kotlin are the most popular frameworks for creating TDs. They also show that TDs are used to replace both classes within the app and external dependencies, that developers use different types of TDs, and that TDs can introduce test smells and even errors in the test code.

Our results motivate further research in this area, justify the development of techniques that can support developers in creating and maintaining TDs, and identify several directions for future work. As a first step, we will present our results to Android developers to gather their feedback, confirm or refuse our findings, and gain further insights. We will also perform additional studies focused on the coupling between apps and their external dependencies to develop analysis techniques that can help identify which classes should be replaced by TDs and which interactions between internal and external code should be mocked and verified. A complementary line of research we will pursue involves the development of techniques for automatically or semi-automatically generating stubs and mocks given a set of relevant classes and interactions. Finally, we will keep performing empirical studies to confirm our results and validate the new techniques we define.

REFERENCES

[1] K. Taneja, Y. Zhang, and T. Xie, "MODA: automated test generation for database applications via mock objects," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 289–292.

[2] A. Arcuri, G. Fraser, and R. Just, "Private api access and functional mocking in automated unit test generation," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2017, pp. 126–137.

[3] T. Bhagya, J. Dietrich, and H. W. Guesgen, "Generating mock skeletons for lightweight web-service testing," in *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*. IEEE, 2019, pp. 181–188.

[4] N. Alshahwan, Y. Jia, K. Lakhotia, G. Fraser, D. Shuler, and P. Tonella, "AUTOMOCK: automated synthesis of a mock environment for test case generation," in *Practical Software Testing: Tool Automation and Human Factors, 14.03. - 19.03.2010*, 2010.

[5] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019, p. 204–215.

[6] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10.

[7] M. L. Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 613–622.

[8] L. Cruz, R. Abreu, and D. Lo, "To the attention of mobile software developers: guess what, test your app!" *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2438–2468, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09701-0

[9] F. Pecorelli, G. Catolino, F. Ferrucci, A. D. Lucia, and F. Palomba, "Testing of mobile applications in the wild: A large-scale empirical study on android apps," in *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 296–307.

[10] J. W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source android apps: A large-scale empirical study," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1078–1089.

[11] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 534–538. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00062

[12] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1461–1498, 2019.

[13] G. Pereira and A. Hora, "Assessing mock classes: An empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 453–463.

[14] H. Zhu, L. Wei, M. Wen, Y. Liu, S.-C. Cheung, Q. Sheng, and C. Zhou, "Mocksniffer: Characterizing and recommending mocking decisions for unit tests," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 436–447.

[15] M. Fazzini, C. Choi, J. M. Copia, G. Lee, Y. Kakehi, A. Gorla, and A. Orso. (2022, Feb.) An artifact for the article: "use of test doubles in android testing: An in-depth investigation". [Online]. Available: https://doi.org/10.5281/zenodo.6000372

[16] (2021, Apr.) Fundamentals of testing. [Online]. Available: https://developer.android.com/training/testing/fundamentals

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

[18] M. Fowler. (2021, Apr.) Testdouble. [Online]. Available: https://martinfowler.com/bliki/TestDouble.html

[19] (2021, Apr.) Mockito. [Online]. Available: https://site.mockito.org

[20] (2021, Apr.) Mockito-kotlin. [Online]. Available: https://github.com/mockito/mockito-kotlin

[21] (2021, Apr.) Powermock. [Online]. Available: https://powermock.github.io

[22] (2021, Apr.) Okhttp. [Online]. Available: https://square.github.io/okhttp

[23] (2021, Apr.) Retrofit. [Online]. Available: https://square.github.io/retrofit

[24] (2021, Apr.) Android test mock. [Online]. Available: https://developer.android.com/reference/android/test/mock/package-summary

[25] (2021, Apr.) Mockito api. [Online]. Available: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

[26] (2021, Apr.) Google play. [Online]. Available: https://play.google.com/store

[27] (2021, Apr.) Javaparser. [Online]. Available: https://javaparser.org

[28] (2021, Apr.) ktlint. [Online]. Available: https://github.com/pinterest/ktlint

[29] (2021, Apr.) Ankidroid. [Online]. Available: https://play.google.com/store/apps/details?id=com.ichi2.anki

[30] (2021, Apr.) Ankidroid github. [Online]. Available: https://github.com/ankidroid/Anki-Android

[31] (2021, Apr.) F-droid. [Online]. Available: https://f-droid.org/en

[32] (2021, Apr.) App manifest overview. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro

[33] (2021, Apr.) Easymock. [Online]. Available: https://easymock.org

[34] (2021, Apr.) jmock. [Online]. Available: http://jmock.org

[35] (2021, Apr.) Mockk. [Online]. Available: https://mockk.io

[36] (2021, Apr.) Mockserver. [Online]. Available: https://www.mock-server.com

[37] (2020, Apr.) Robolectric. [Online]. Available: http://robolectric.org

[38] (2021, Apr.) Rxandroidble. [Online]. Available: https://github.com/Polidea/RxAndroidBle

[39] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, p. 119–130.

[40] T. Wendland, J. Sun, J. Mahmud, S. M. H. Mansur, S. Huang, K. Moran, J. Rubin, and M. Fazzini, "Andror2: A dataset of manually-reproduced bug reports for android apps," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 600–604.

[41] J. Johnson, J. Mahmud, T. Wendland, K. Moran, J. Rubin, and M. Fazzini, "An empirical investigation into the reproduction of bug reports for android apps," in *Proceedings of the 29th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE Computer Society, 2022.

[42] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[43] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative data analysis: A methods sourcebook*. Sage publications, 2018.

[44] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. USA: IBM Corp., 2010, p. 214–224.

[45] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, p. 402–412.

[46] K. Krippendorff, "Reliability in content analysis: Some common misconceptions and recommendations," *Human communication research*, vol. 30, no. 3, pp. 411–433, 2004.

[47] ——, *Content analysis: An introduction to its methodology*. Sage publications, 2004.

[48] (2021, Apr.) Mockitohint. [Online]. Available: https://javadoc.io/static/org.mockito/mockito-core/3.2.4/org/mockito/quality/MockitoHint.html

[49] (2021, Apr.) Mockito strictness documentation. [Online]. Available: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/quality/Strictness.html

[50] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *2014 14th International Conference on Quality Software*, 2014, pp. 127–132.

[51] B. G. Mateus and M. Martinez, "An empirical study on quality of android applications written in kotlin language," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3356–3393, 2019.

[52] V. Oliveira, L. Teixeira, and F. Ebert, "On the adoption of kotlin on android development: A triangulation study," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 206–216.

[53] (2021, Apr.) Update on kotlin for android. [Online]. Available: https://android-developers.googleblog.com/2017/11/update-on-kotlin-for-android.html

[54] (2021, Apr.) String resources. [Online]. Available: https://developer.android.com/guide/topics/resources/string-resource

[55] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. USA: IBM Corp., 2019, p. 193–202.

[56] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 71–82.

[57] H. Wang and Y. Guo, "Understanding third-party libraries in mobile app analysis," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. IEEE Press, 2017, p. 515–516.

[58] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 70–79.

[59] (2021, Dec.) Build local unit testsk. [Online]. Available: https://developer.android.com/training/testing/unit-testing/local-unit-tests

[60] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2006, p. 253–264.

[61] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software*

Engineering, vol. 35, no. 1, pp. 29–45, 2009.
[62] A. Kampmann and A. Zeller, "Carving parameterized unit tests," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 248–249.
[63] (2020, Apr.) Robolectric. [Online]. Available: http://robolectric.org/extending
[64] F. Trautsch and J. Grabowski, "Are there any unit tests? an empirical study on unit testing in open source python projects," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 207–218.
[65] M. Islam and C. Csallner, "Generating test cases for programs that are coded against interfaces and annotations," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, pp. 21:1–21:38, 2014.
[66] A. Arcuri, G. Fraser, and J. P. Galeotti, "Generating TCP/UDP network data for automated unit test generation," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 155–165.
[67] L. Gazzola, M. Goldstein, L. Mariani, I. Segall, and L. Ussi, "Automatic ex-vivo regression testing of microservices," in *AST@ICSE 2020: IEEE/ACM 1st International Conference on Automation of Software Test, Seoul, Republic of Korea, 15-16 July, 2020*. ACM, 2020, pp. 11–20.
[68] P. Zhang and S. G. Elbaum, "Amplifying tests to validate exception handling code: An extended study in the mobile application domain," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 32:1–32:28, 2014.
[69] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, "Identifying java calls in native code via binary scanning," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 388–400.
[70] L. Brutschy, P. Ferrara, O. Tripp, and M. Pistoia, "Shamdroid: gracefully degrading functionality in the presence of limited resource access," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, J. Aldrich and P. Eugster, Eds. ACM, 2015, pp. 316–331.
[71] B. Mariano, J. Reese, S. Xu, T. Nguyen, X. Qiu, J. S. Foster, and A. Solar-Lezama, "Program synthesis with algebraic library specifications," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 132:1–132:25, 2019.
[72] X. Wang, L. Xiao, T. Yu, A. Woepse, and S. Wong, "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*. New York, NY, USA: Association for Computing Machinery, 2021, p. 540–552.
[73] M. Fazzini, A. Gorla, and A. Orso, "A framework for automated test mocking of mobile apps," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1204–1208.