Automated Tests for Mobile Games: an Experience Report

Gabriel Lovreto, Andre T. Endo, Paulo Nardi
Department of Computing
Federal University of Technology - Parana
Cornelio Procopio, Brazil
gabriel.lovreto@gmail.com, {andreendo,paulonardi}@utfpr.edu.br

Vinicius H. S. Durelli
Department of Computer Science
Federal University of Sao Joao del Rei
Sao Joao del Rei, Brazil
durelli@ufsj.edu.br

***Abstract*—As mobile gaming is an ever-growing, competitive and profitable market, there has been an increasing demand for better quality in video game software. While manual testing is still a common practice among mobile game developers, some repetitive and error-prone tasks could benefit from test automation. For instance, test scripts that perform sanity checks of the proper functioning of a mobile game would be desirable in an ecosystem with constant hotfixes and updates, as well as a diverse set of configurations (e.g., device hardware, screensizes, and platforms). In this context, this paper reports an experience on developing automated test scripts for mobile games. To this end, we randomly selected 16 mobile games, from different genres, among the popular ones from the Google Play Store. For each game, test scripts were developed using the Appium testing framework and the OpenCV library. Based on our results, we provide an in-depth discussion on the challenges and lessons learned.**

***Keywords*-Mobile Apps; Software Testing; Mobile Games; Video Game Software; Test Cases**

I. Introduction

The videogame market is growing at a rapid rate in what has already become a billionaire business. In particular, games for mobile devices have received special attention. Newzoo's global market research [1] reports a 29% market share for phone games and 10% for tablets in 2016, while consoles and PC had around 32% and 30%, respectively. The same source predicts that the share of phone and tablet games will be responsible for 50% of the market. The videogames developed to run in such devices as smartphones and tablets are known as *mobile games*. As mobile games are also apps, they are distributed to the end-users by means of well-known app stores like Google Play (for Android devices) and Apple App Store (for iOS devices). Besides distribution, these app stores also provide means for players to rate the game, write reviews, and receive updates/bug-fixes. This ecosystem brings a high level of competition (e.g., by the presence of game clones) and a constant demand for better quality and user experience. Moreover, the presence of bugs can negatively impact the game's rating score and, as a consequence, affect its downloads [2], [3].

When developing games, as with any software, testing is one of the important aspects involved in guaranteeing product quality [4]. Game testing is mostly manual and, as such, depends on several human testers. This scenario presents not only high time and human resource costs, but also introduces the potential for human error. This is corroborated by Whittaker [5], when he argues that manually applying test scenarios is labor-intensive and error-prone.

As games get more popular and complex, testing them becomes a key challenge. Lin et al. [6] argue that even the most popular games in the market show signs of lack of proper testing. When Alemm et al. [4] uncovered that most companies still follow manual testing procedures with disregard for formal testing methods, they also pointed out that proper testing was one of the most important factors in creating a successful game. Khalid et al. [7] argue that mobile game development is a very complicated area in regard to platforms (especially Android), as there can be up to a hundred or more different devices running, or trying to run a game. Manual testing for that many devices is not only a very difficult task but also costly.

There is an extensive body of knowledge on mobile app testing; see Zein et al. [8] for a comprehensive literature review. While there exists a particular interest in test automation [9]–[11], such initiatives are focused on general-purpose apps and not mobile games. If automated test scripts were to be introduced into the mobile gaming development, they could create an economy, or at least a better distribution, of human resources. Instead of relying on several human testers for repetitive technical tasks, test scripts could be developed to automate such tests. This would, in turn, let the testers be employed to human experience-related tests, such as game balance[1] testing. While some tests would benefit from an automated script (avoiding human error), the player experience would be too difficult to be properly tested automatically as the game can present too many variables and not one exact "correct" result. Therefore, automated tests can bring several benefits, like better verification of hotfixes and updates [6], and cheaper sanity checks[2] in several configurations (varying device hardware, screensize, platform, and so on) [7].

[1]Essentially, how "fair" a game is to its players.

[2]Sanity tests are employed to exercise parts of the application under test to determine its basic and proper functioning.

This paper reports an experience on automating test cases for mobile games. We randomly selected 16 free games among the most popular ones in the Google Play Store. In order to have a diverse set, each game is from a different genre. For each game, we followed a pragmatic procedure to design, configure, code, and run test scripts. Such procedure is based on Python scripts, the GUI testing framework Appium, and the computer vision library OpenCV. Finally, we also elicit and discuss the challenges involved, as well as the lessons learned.

This paper is organized as follows: Section II presents the background. Section III discusses the related work. Section IV reports the study configuration and adopted tools. Section V analyzes the results and presents the lessons learned. Finally, Section VI makes the concluding remarks and sketches future work.

II. Background

Mobile gaming, according to Novak [12], can be traced all the way back to the first portable consoles released in the market by Mattel. As mobile devices like smartphones and tablets have become increasingly popular, they have overtaken portable gaming consoles in Market share. Hsiao et al. [13] mention the complexity of current mobile games and the growth of in-app purchases, pointing out several factors that influence a player's loyalty to the game. While several classifications have been proposed in the literature [12], mobile games are usually classified as the genres provided by the app stores; some of the genres defined by Google Play are [14]: action, adventure, arcade, board, cards, casino, casual, educational, music, racing, RPG, simulation, sports, strategy, trivia/quizzes, and words.

As any kind of software, companies strive to produce high quality games and use a variety of techniques to cope with the complexity involved in developing games. Among them, ***Software Testing*** is essential in ensuring that quality standards are met and is valuable to decrease the overall development and maintenance efforts. Software testing can be summed up as *"the process of executing a program with the intent of finding errors"* [15]. Myers et al. [15] also argue that testing is more than just making sure a program runs as intended – it is also about adding value to the final product by virtue of raising its reliability and quality. By finding errors through program execution, developers are increasing the inherent value of a given game.

A central element in software testing is the test case. A test case represents a usage scenario in which the inputs and expected outputs are defined, as well as the execution conditions. In this research, test cases are at system level and focused on graphical user interfaces (GUIs). Such test cases can be executed manually or in an automatic way. As for manual testing, a technique usually applied in games is exploratory testing [4]. Exploratory testing is the combination of learning, test design, and execution and has a session-based management [16]. Concerning automation, scripts can be coded to execute the test cases automatically in the game under test.

In this research, we implemented the test scripts in Python using the `Appium` framework. `Appium` [17] is an open-source framework to develop automated tests for mobile apps. We selected it to test mobile games based on the following reasons. First, it allows black-box testing without requiring any internal modifications to the app. Second, the developer can write test scripts using the well-known WebDriver API for different mobile platforms (e.g., Android and iOS). Finally, `Appium` supports several programming languages like C#, Java, Python, and so on.

`Appium` works through a client/server architecture, as it uses a web server that receives commands from a remote client and executes them on a mobile device. Figure 1 gives an overview of its architecture. The Appium Server and test scripts can be executed in different machines. The target device could be an emulator or a physical device.

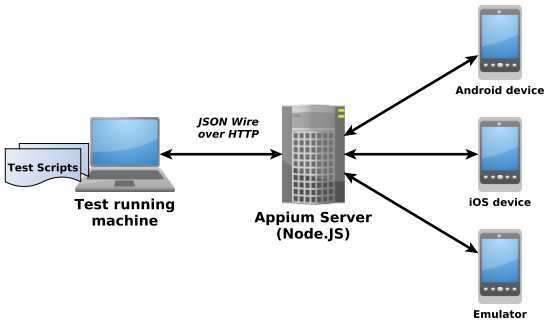

Figure 1. `Appium` architecture, adapted from [18].

In the following, we discuss some code snippets to show how to set up a Python client and run a simple test on an Android device. Figure 2 shows the main configuration steps. Lines 1-4 import libraries that will be used in the tests, namely specific Operational System functions (`os`), sleep instructions (`time`), unit testing framework (`unittest`), and class "webdriver" from the `Appium`. Lines 8-15 are the settings that will be sent to the Appium server so it knows what platform, device and app to run. Lines 15-17 declare variable "driver" and connects to the Appium server hosted at "http://localhost:4723/wd/hub".

Figure 3 shows a script to automate a test case. Line 1 declares a function that automates a test case. Lines 2-3 search, through the app's current GUI elements, for an element with the accessibility ID 'Graphics'. Then, it clicks in it in Line 4. GUI elements are also retrieved in Lines 5-6, 11-12, 15-17, and 20-22. Expected outputs are verified in assertions made in Lines 7, 13, and 18. Line 9 performs a "back" event, essentially emulating the pressing of Android's

```
import os
from time import sleep
import unittest
from appium import webdriver

class SimpleAndroidTests(unittest.TestCase):
  def setUp(self):
    desired_caps = {}
    desired_caps['platformName'] = 'Android'
    desired_caps['platformVersion'] = '4.2'
    desired_caps['deviceName'] = 'Android
        Emulator'
    desired_caps['app'] = PATH(
         '../../../sample-code/apps/ApiDemos/' +
         'bin/ApiDemos-debug.apk' )
    self.driver = webdriver.Remote(
              'http://localhost:4723/wd/hub',
              desired_caps)
```

Figure 2. Python script (setup).

back button.

```
def test_find_elements(self):
  el = self.driver
    .find_element_by_accessibility_id('Graphics')
  el.click()
  el = self.driver
    .find_element_by_accessibility_id('Arcs')
  self.assertIsNotNone(el)

  self.driver.back()

  el = self.driver
    .find_element_by_accessibility_id('App')
  self.assertIsNotNone(el)

  els = self.driver
    .find_elements_by_android_uiautomator
      ('new UiSelector().clickable(true)')
  self.assertGreaterEqual(12, len(els))

  self.driver
    .find_element_by_android_uiautomator
            ('text("API Demos")')
```

Figure 3. Python script (test case).

III. Related Work

Alemm et al. [4] elicit success factors for mobile game development, in which different types of testing are discussed. By surveying several companies in the mobile games market, the authors noted that systematic and automated testing was only a fraction, being ad-hoc and exploratory tests predominant. Several papers in the literature have focused on automated testing for games. In general, researchers and practitioners hope to achieve faster and more scalable tests, while reducing the human effort on repetitive and error-prone tasks.

Iftikhar et al. [19] describe a UML-based model for automated game testing. In particular, the authors aimed to test platform games, dealing with test case generation, oracle generation, and test case execution. Parts of the game are modeled as a state machine; from such a model, test cases are generated by traversing the state machine. The oracle is based on system events, especially the ones created as response to user events. A limitation is that it requires an extra effort for generating the oracles and test cases (i.e., the model), but it has potential to reduce human resources in the test execution phase.

Lin et al. [6] investigate urgent patches/updates of popular games. They define an update/patch as urgent when released outside of the usual schedule or mentioned explicitly as a hotfix. The authors gathered data from several different sources for the 50 most popular games on Steam. Not all hotfixes or urgent patches are released to fix functional bugs or technical problems; many aim to fix the game balance and change its rules to create a better playing experience. Among the functional bugs, they are usually related to crashes and visual/graphic flaws.

Khalid et al. [7] analyze the most useful reviews for the 99 most popular free Android games. The results show that reviews came from users using from 38 to 132 different devices. Such approach could be an important tool in finding out which device, or devices, generates the lower score reviews for certain game genres.

Although there exist initiatives for games in general (like [19]), the testing of mobile games has not received attention from academia. To the best of our knowledge, this is the first study that aims to investigate the challenges on developing automated tests for mobile games. We hope to foster future initiatives on more replicable and automated tests for mobile games.

IV. Study Configuration

This study aims to investigate the viability and efficiency of using automated test scripts for mobile games. We selected `Python` as programming language for coding the test scripts run by `Appium`. We also used `UIAutomator Viewer` to inspect UI elements in compatible games and `OpenCV` [20] for image recognition and processing.

The `Appium` API allows to execute the tests in the connected device, capturing images, performing touch actions – such as taps or slides –, and finding elements by the resource-ids (whenever a game was compatible with `UIAutomator`). `Appium` executes as a local server that connects to the target device, plugged to the computer through the USB port. Tests were written as `Python` scripts, employing specific libraries for `Appium` and `OpenCV`.

`UIAutomator Viewer` was used to inspect the XML hierarchy of UI elements [21]. From such XML, it is possible to select special ids used by the Appium API to interact with UI elements. While this strategy is common in apps in general, most mobile games do not provide an XML hierarchy of their UI elements. Therefore, we adopted `OpenCV`

template matching to find elements by their similarity with previously-captured images. The templates are manually-edited images extracted from screenshots of the game under test; they represent elements that we intend to interact with like buttons, characters, items, enemies, and so on. During the test execution, screenshots of the game are captured and the templates' position are determined by processing images on-the-fly. The `OpenCV` template matching is done by "sliding" the template over the image, comparing the template with the part of the image underneath it [22]. We configure the similarity threshold for each game tested.

For this study, we developed a class[3] to reduce duplicated code and improve readability of the test scripts. This class has three methods: *(i)* finding matches between two images (a template and a screenshot), *(ii)* waiting for loading screens, and *(iii)* skipping ads.

First, we selected the 20 most popular free games for each genre listed in the Google Play Store[4]. For each genre, one game was randomly chosen as subject. As we did not intend to test player to player interaction, multiplayer-only games were discarded. The study was conducted with 16 mobile games. All games were downloaded through the North American Google Play Store and tested in a LG G Pad device running Android version 5.0.2.

Figure 4 shows how we conducted the study for each game. Initially, we performed exploratory tests in order to understand the game's mechanics. Then, we tried to design two types of test cases, one focused on performing actions that simulate someone playing (called "game tests"), and the other aimed to walk through the game's menus (called "menu tests").

After specifying the test cases, we started the steps to write a test automation script. First, we checked if the game has an XML hierarchy of UI elements (i.e., compatible with UIAutomator). If true, `Automator Viewer` was used to analyze the screens and extract the UI elements' ids. Otherwise, screenshots of the game were captured and templates created. Such resources were organized to be accessed in the scripts.

A first version of the test script was then coded and executed. As an iterative process, we went back to re-arrange the test cases and run again if the results were unexpected. Otherwise, the results were collected and analyzed and the workflow started again with the next mobile game.

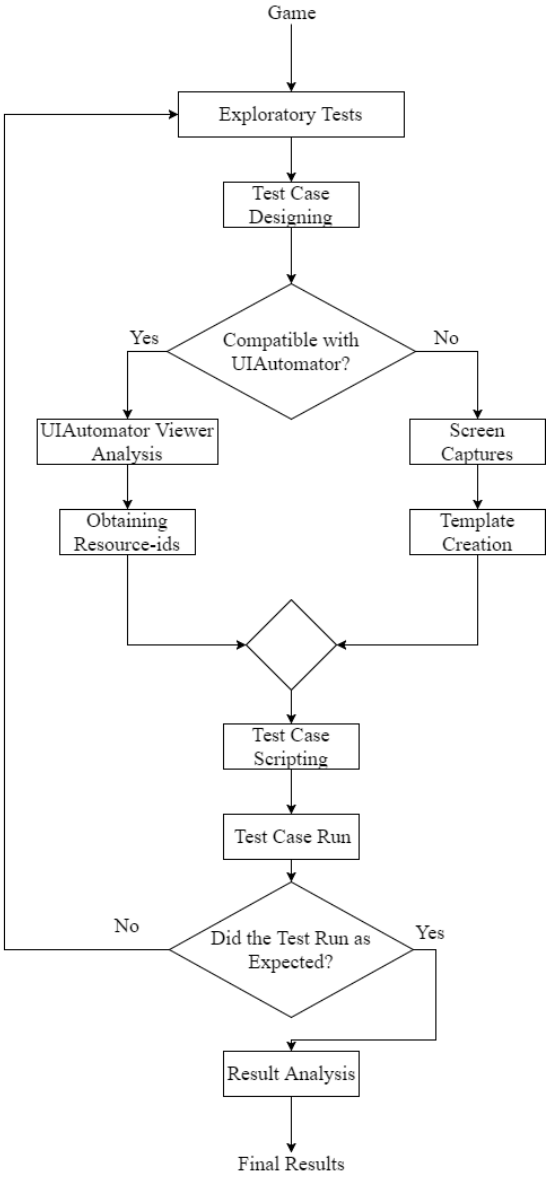

Figure 4. Workflow of the study.

V. Results and Lessons Learned

Table I shows the 16 selected games, their genre, number of downloads and average star rating[5]. Notice that the games have many players, most of them have more than 1 million downloads. `Ludo King` has over 10 million downloads, while `titans subway go` has the fewest number of downloads (less than 500,000). The sample has 4.3 stars on average; `titans subway go` has the smallest star rating (3.8), and `Piano Kids - Music Songs` and `Drogo` have the highest values (4.6). Overall, the sample contains popular games from different genres with a high star rating.

Figure 5 shows how many games were or not compatible with UIAutomator for testing without templates. Only two (approximately 12.5%) were compatible with UIAutomator and 14 games (approximately 87.5%) were not. Most games are not compatible with UIAutomator because their GUIs worked "outside" of Android's layout hierarchy. This means

[3]The code is available at https://github.com/tyrus1235/Appium-OpenCV-MobileTesting

[4]As of July 31st, 2017

[5]Data was collected from Play Store in October 2017.

Table I
LIST OF SELECTED MOBILE GAMES; DATA OBTAINED FROM GOOGLE PLAY.

Game Name	Game Genre	Number of Downloads	Average Star Rating (0 to 5)
Shoot Hunter-Gun Killer	Action	10 000 000 - 50 000 000	4.3
titans subway go	Adventure	100 000 - 500 000	3.8
Sonic Boom	Arcade	10 000 000 - 50 000 000	4.4
Ludo King	Board	50 000 000 - 100 000 000	4.4
Spider	Card	10 000 000 - 50 000 000	4.0
Bingo Pop	Casino	10 000 000 - 50 000 000	4.2
Cookie Crush Match 3	Casual	10 000 000 - 50 000 000	4.4
Math Games	Educational	5 000 000 - 10 000 000	4.2
Piano Kids - Music Songs	Music	1 000 000 - 5 000 000	4.6
Moto Rider GO: Highway Traffic	Racing	10 000 000 - 50 000 000	4.3
Pony Sisters - Baby Horse Care	RPG	1 000 000 - 5 000 000	4.3
Drogo	Simulation	1 000 000 - 5 000 000	4.6
Flip Diving	Sports	10 000 000 - 50 000 000	4.5
Art of Conquest	Strategy	1 000 000 - 5 000 000	4.3
QuizUp	Trivia	10 000 000 - 50 000 000	4.3
Word Search	Word	1 000 000 - 5 000 000	4.3

that the game screen would be shown inside a single “pane” with no additional internal objects while ads and such would have their own, separate hierarchy.

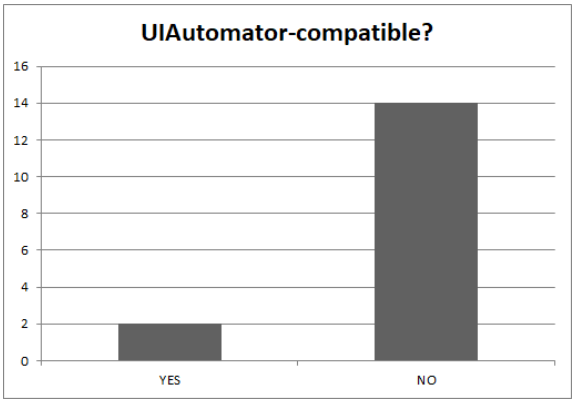

Figure 5. Number of games compatible with UIAutomator.

Table II summarizes the main results with respect to the test cases designed. For each game, it shows the genre, type of test case, script’s number of lines of code (Script LoC), number of test steps, number of templates (#Temp.), number of successful test executions, number of failed test executions, and the average execution time.

Observe that not all games had the two types of test case since there were games without menus, while others had almost no gameplay. These games are Moto Rider GO, Pony Sisters, Drogo, and Art of Conquest. As for Moto Rider GO, it forces a first-time player to go through an extensive tutorial. This tutorial includes gameplay and menu interaction, so one single test case was designed for both. Pony Sisters did not have any traditional (or elaborated, for that matter) menus, so only its gameplay was tested. Both Drogo and Art of Conquest did not have their gameplay tested - albeit for opposite reasons. While Drogo’s gameplay is too simplistic to design a relevant test case, Art of Conquest is too complex and would take an prohibitive amount of time to complete (not to mention including multiplayer interaction, which is beyond this study’s scope).

Concerning script LoC, Moto Rider GO had a script with the highest LoC. It has only one test script that encompasses both gameplay and menu interaction (for reasons mentioned afore). Following, there is the gameplay test case for Ludo King. Initially, the game has many configuration steps and screens and then it mostly consists of touching the die to roll it and then touching a piece to move it. On the other hand, the script with the lowest LoC is the gameplay test for Spider. Spider is a simple card game that starts right into the game itself. Since testing was mostly focused on whether all cards from the player’s hand could be dealt, the script is mainly a loop of picking the top card and seeing the results.

For column #Test Steps in Table II, notice that some games feature x mandatory test steps and y optional steps, represented as $x(y)$. Optional steps represent behavior that is not guaranteed to happen for every execution, e.g., daily rewards, requests for permissions, and so on. Since this uncertainty is the intended behavior of some games, we dealt with it in the test scripts. The gameplay test case for Bingo Pop has the greatest number of test steps (32). Part of the test case involves tapping each and every possible position on both bingo cards during the game. Similarly, Piano Kids has a test case with 27 steps. Most of the games had tests with steps in the 7-9 range. Math Games’ gameplay test case has the fewest number of steps and is an interesting case; the only action executed is tapping the ”Play” button. The gameplay itself is tested by reading the question (in this case, a simple sum of two numbers), solving it and trying to find the correct answer amongst the ones presented. It is

Table II
TEST CASE RESULTS.

Game Name	Game Genre	TC Type	Script LoC	#Test Steps	#Temp.	Succesful Tests	Failed Tests	Average Exec. Time
Shoot Hunter-Gun Killer	Action	Menu Game	96 107	3 3(4)	16	86.67% 83.33%	13.33% 16.67%	51.92 s 109.72 s
titans subway go	Adventure	Menu Game	82 113	2 6	8	100% 100%	0% 0%	38.31 s 43.59 s
Sonic Boom	Arcade	Menu Game	165 152	6(8) 7(10)	13	6.67% 73.33%	93.33% 26.67%	73.81 s 92.18 s
Ludo King	Board	Menu Game	168 199	7(8) 12(13)	19	0% 0%	100% 100%	48.38 s 44.49 s
Spider	Card	Menu Game	86 48	3 5	5	100% 96.67%	0% 3.33%	35.36 s 40.72 s
Bingo Pop	Casino	Menu Game	142 172	5 32	21	0% 0%	100% 100%	74.03 s 72.70 s
Cookie Crush Match 3	Casual	Menu Game	75 90	2 4	7	100% 100%	0% 0%	54.60 s 64.61 s
Math Games	Educational	Menu Game	60 62	2 1	N/A	100% 100%	0% 0%	25.10 s 48.14 s
Piano Kids – Music Songs	Music	Menu Game	95 138	3 27	14	100% 23.33%	0% 76.67%	43.42 s 46.64 s
Moto Rider GO: Highway Traffic	Racing	Menu Game	293	16(17)	19	0%	100%	72.60 s
Pony Sisters – Baby Horse Care	RPG	Menu Game	N/A 100	N/A 4	9	N/A 0%	N/A 100%	N/A 41.49 s
Drogo	Simulation	Menu Game	157 N/A	7(8) N/A	11	10% N/A	90% N/A	69.80 s N/A
Flip Diving	Sports	Menu Game	91 83	3 5	8	30% 13.33%	70% 86.67%	59.58 s 57.04 s
Art of Conquest	Strategy	Menu Game	125 N/A	5 N/A	8	0% N/A	100% N/A	49.86 s N/A
QuizUp	Trivia	Menu Game	126 150	5(7) 9	N/A	100% 23.33%	0% 76.67%	47.57 s 67 s
Word Search	Word	Menu Game	114 87	4(5) 2(4)	11	96.67% 10%	3.33% 90%	31.76 s 53.02 s

only possible to read the question and answers because the game was compatible with UIAutomator and, as such, the text inside its graphical elements was accessible through a query.

Column "#Temp." of Table II represents how many template images were created and used for each game. Since templates could often be reused between test cases of the same game, we counted only the total number of templates. The game with most templates was Bingo Pop. This was due to the amount of loading screens the game had – requiring unique loading templates for the test case to wait them out, since they all looked different – and the amount of menus and pop-ups that stood between the start and the gameplay or the settings menu. The game with the fewest number of templates was Spider; it does not have any loading screens, has a simplistic GUI and starts right at the gameplay (with the settings menu just a couple of taps away).

The last three columns of Table II result from 30 executions for each test script. The varied success and failure rates (in columns "Successful tests " and "Failed tests") shed some light on a troublesome characteristic of mobile game testing: flaky tests. In general we expect test cases to be replicable, so they need to be deterministic. Therefore, given the exact same context, every execution of the test script should have the same result – be it failure or success. For half the games observed in this study, their tests were not deterministic – they were flaky tests. As Luo et al. [23] comment about them, flaky tests undermine testing, as test results become unreliable. As for why mobile games had flaky tests, some factors seem to be the culprits: the random nature of ads and the limitations of template matching.

Most of the test cases were deterministic: around 57.14% (16 out of 28). Looking at the TC type, menu-based tests had the exact same rate of deterministic tests as all types together (approximately 57.14%, or 8 out of 14), while gameplay tests had a lower rate of deterministic tests – approximately 46.15% (or 6 out of 13). Since gameplay tests generally present more test steps and usually have a variable context, this is expected. Also, animations and other graphical features that vary scale, rotation or shape of an element do not work well with template matching, as it requires such characteristics to be constant for matching.

Moving on to the "average execution time" column

of II, the longest-running test was Shoot Hunter-Gun Killer's gameplay test. This was due to several intermediary screens between the game launch and the gameplay start. There are three loading screens (each one takes at least two seconds) as well as an ad screen (5 seconds, at the least). Math Games had the quickest test. Two factors seem to be the reasons: *(i)* it did not use image recognition – meaning less time spent doing template matching and analyzing its results – and *(ii)* it has a very simple menu structure, meaning less screens and loading times to test. Following Math Games's and QuizUp's tests, the quickest test that uses template matching was Spider's menu test. Notice that QuizUp's test had a mandatory "registration" with several steps and screens. This affected its test execution time so much that, even though it did not use image recognition, was still slower than many other tests (which used OpenCV template matching). In general, menu tests are generally quicker than gameplay tests. This was expected since most games had more test steps in their gameplay tests than their menu tests. In other words, there are more screens, more actions and more loading times, increasing the average test execution time.

Table III shows the approximate effort spent (in hours/tester) for the main tasks of the study per game. Exploratory analysis and template creation were the less costly tasks with 1.7h and 1.5h on average, respectively. Test case design and script coding took longer, 1.9h and 3.9h on average. In total, an average of almost nine hours (8.8) has been spent in each mobile game.

Notice that a lot of effort has been spent with Moto Rider GO: Highway Traffic and Bingo Pop; such games are complex and demanded several hours to have functioning test scripts. They present a large number of graphical elements (some of those with variable size, shape or rotation) and several different menu screens. On the other hand, the tasks for Spider, Math Games and Flip Diving were the quickest (4h). Concerning Math Games, the tasks benefited from the absence of template creation (as it was a game compatible with UIAutomator) and from its simple GUI (no animations, few screens and loading times). As for Spider, it has no loading screen, at most three screens using actually one screen for the whole gameplay test, no animations or other complex graphical elements. Flip Diving diverges from this pattern; it is not a simple game, has loading screens, and some GUI elements are complex. Yet, its gameplay and menus were not complex and the template matching was not hard to set up.

A. Lessons Learned

Testing mobile games brought out many challenges as we observed in our study. Some of those were overcome by employing pragmatic solutions, while others remained untackled for the reasons discussed as follows.

Recognition of Graphical Elements: Due to how template matching works, some graphical elements proved rather difficult to work with. For example, in Shoot Hunter-Gun Killer, the "shoot" button was semi-transparent - this meant that whatever was behind the button had to match the template in order for the button itself to be found. Another example was in Bingo Pop, where several buttons had animations that changed scale, rotation or appearance as a whole. Since the template would be based on a single frame of the said animation, the only choice for a good image recognition (i.e., one that would not need to have a low threshold, risking finding more elements than expected) was if the game screen was captured in the exact same frame of animation. Naturally, this causes failed test cases since certain buttons were not recognized. Other games faced this issue, including Pony Sisters, Piano Kids, Moto Rider GO, and Ludo King.

The last issue was uncovered while testing Piano Kids' gameplay. In its musical keyboard screen, there are two "Do" keys only differentiated by their colors. Since template matching does not take colors into account, when trying to find the first "Do" key, the code either found two keys or (when its threshold was risen to almost maximum) just the second key. This could be a problem in games that feature the same graphical element with different meanings and/or different colors.

Loading and Ads: Since the games are free, all of them featured ads in one form or another. While ad banners did not influence testing in any perceivable way, ad screens with required inputs (in general, tapping the close button) showed to be a big challenge. This is mostly due to the random nature of such ad screens – be it their timing (sometimes they show up, sometimes they do not) or their GUI (different-looking close buttons meant difficulty in choosing the correct templates). Because of these ads, several test cases' successul executions were based on chance rather than deterministic states.

As for loading-related issues, some games had varying loading times. This could cause the script to skip a loading screen simply because it had not finished rendering. In games like Art of Conquest, several loading screens showed up successively, so the OpenCV's template matching was not fast enough to analyze whether a loading screen was still ongoing before it finished. Although generally not a critical issue for the test results (except when a loading screen would be "skipped" by the script right before a GUI interaction, causing the test to incorrectly fail), it does affect general test performance and execution time.

Active Player Events: Most games and tests only required taps to navigate in menus or interact with the gameplay. Those were simple to emulate as Appium features a "tap" function that accepts screen coordinates as parameter. This is important, as most other Appium functions require an element object (obtained through its own queries) as

Table III
EFFORT SPENT PER TASK.

Game Name	Exploratory Analysis	Test Case Design	Template Creation	Script Coding	Total
Shoot Hunter-Gun Killer	1 hour	2 hours	3 hours	6 hours	12 hours
titans subway go	1 hour	1 hour	1 hour	3 hours	6 hours
Sonic Boom	2 hours	2 hours	1 hour	8 hours	13 hours
Ludo King	2 hours	3 hours	1 hour	6 hours	12 hours
Spider	1 hour	1 hour	1 hour	1 hour	4 hours
Bingo Pop	3 hours	2 hours	4 hours	6 hours	15 hours
Cookie Crush Match 3	1 hour	1 hour	1 hour	2 hours	5 hours
Math Games	1 hour	1 hour	-	2 hours	4 hours
Piano Kids - Music Songs	2 hours	2 hours	1 hour	3 hours	8 hours
Moto Rider GO: Highway Traffic	4 hours	7 hours	3 hours	5 hours	19 hours
Pony Sisters - Baby Horse Care	1 hour	2 hours	1 hour	2 hours	6 hours
Drogo	1 hour	1 hour	1 hour	2 hours	5 hours
Flip Diving	1 hour	1 hour	1 hour	1 hour	4 hours
Art of Conquest	4 hours	3 hours	1 hour	6 hours	14 hours
QuizUp	1 hour	1 hour	-	5 hours	7 hours
Word Search	1 hour	1 hour	1 hour	4 hours	7 hours
Average	1.7 hours	1.9 hours	1.5 hours	3.9 hours	8.8 hours

parameter and OpenCV's template matching could only return coordinates.

Nevertheless, some games required swipe or drag-and-drop events to interact with their GUIs or gameplay. For those two types of interaction, Appium's "swipe" function was essential. It accepted as parameters coordinates for starting and end points, as well as the speed of the action. For instance, when moving the "Volume" slider all the way to the right in Shoot Hunter's menu test, the starting position was the slider knob, the end position was the slider's right tip and the execution time was one second. This effectively emulated a drag-and-drop movement only using coordinates obtained through OpenCV's template matching. For games like Sonic Boom and titans subway go, in-game swipes used the middle of the screen as the starting point and an arbitrary end point either above (for jumping) or left/right (for avoiding obstacles). This was not always accurate (e.g., in titans subway go, the command would not always be recognized as an upwards swipe.

VI. CONCLUDING REMARKS

Automated testing is appealing for mobile games; such a class of games is characterized by being deployed in several devices and platforms. This paper reports an experience on developing automated tests for 16 mobile games. Some factors that heavily influenced the viability (and efficiency) of the test scripts were Appium and OpenCV's performance, how unpredictable the game was, and the template matching limitations. Most test cases did not take more than one minute to execute; therefore, the tools selected showed a reasonable performance, though faster test cases would be desirable. Test flakiness is a challenge due to the unpredictable nature of games and template matching limitations.

In the future, two main issues require further investigation: unpredictability management and image feature recognition function. As for unpredictability, future directions involve configurable test cases and random (monkey) testing. Concerning better image recognition for testing, one could use either keypoint mapping or a more precise feature detection algorithm. OpenCV provides other algorithms that can be tuned and deal with specific features like rotation and scale. These functions could diminish the amount of trial-and-error steps that template matching requires to set up the threshold. In summary, there is a lot of room for improvements since this is a fairly new subject and there exists a lack of significant advances from researchers and practitioners.

REFERENCES

[1] E. McDonald, "The global games market will reach $108.9 billion in 2017 with mobile taking 42%," 2017. [Online]. Available: https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/

[2] M. Harman, Y. Jia, and Y. Zhang, "App store mining and analysis: Msr for app stores," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 108–111.

[3] H.-W. Kim, H.-L. Lee, and S.-J. Choi, "An exploratory study on the determinants of mobile application purchase," *The Journal of Society for e-Business Studies*, vol. 16, no. 4, pp. 173–195, 2011.

[4] S. Alemm, L. F. Capretz, and F. Ahmed, "Critical success factors to improve the game development process from a developer's perspective," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 925–950, 2016.

[5] J. A. Whittaker, "What is software testing? and why is it so hard?" *IEEE software*, vol. 17, no. 1, pp. 70–79, 2000.

[6] D. Lin, C.-P. Bezemer, and A. E. Hassan, "Studying the urgent updates of popular games on the steam platform," *Empirical Software Engineering*, pp. 1–32, 2016.

[7] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: A case study of android game apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 610–620.

[8] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *Journal of Systems and Software*, vol. 117, pp. 334–356, 2016.

[9] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Washington, DC, USA, 2015, pp. 429–440. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.89

[10] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: ACM, 2016, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054

[11] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 245–256. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106298

[12] J. Novak, *Game Development Essentials: An Introduction*, 3rd ed. Cengage Learning, 2011.

[13] K.-L. Hsiao and C.-C. Chen, "What drives in-app purchase intention for mobile games? an examination of perceived values and loyalty," *Electronic Commerce Research and Applications*, vol. 16, pp. 18–29, 2016.

[14] Google, "Google play store," 2018. [Online]. Available: https://play.google.com/store/apps/category/GAME

[15] G. J. Myers, T. Badgett, and C. Sandler, *The Art of Software Testing*, 3rd ed. USA: John Wiley & Sons, 2011.

[16] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," in *Empirical Software Engineering, 2005. 2005 International Symposium on*. IEEE, 2005, pp. 10–pp.

[17] Appium, "Appium: Automation for apps," 2018. [Online]. Available: http://appium.io/introduction.html

[18] RightQA, "An introduction to appium architecture," 2015. [Online]. Available: http://blog.rightqa.com/2015/07/an-introduction-to-appium-architecture.html

[19] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, "An automated model based testing approach for platform games," in *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE, 2015, pp. 426–435.

[20] OpenCV, "Opencv – open source computer vision library," 2018. [Online]. Available: https://opencv.org

[21] A. Developers, "Ui automator," 2018. [Online]. Available: https://developer.android.com/training/testing/ui-automator.html

[22] OpenCV, "Python template matching," 2018. [Online]. Available: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html

[23] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.