Mobile UI Testing

with Xamarin Applications

Presented by Maïlys Terrier

Table of Contents

Introduction

With the increased dependencies on our smartphone devices, mobile application development has been at an all-time high. Mobile development is starting to prioritize cross-platform compatibility with the wide array of mobile devices available today, but with the various screen dimensions and screen resolution, the intended UI developed often does not appear the way the developer expected. Therefore, this mobile development must not only focus on the functionality of the app but also the user experience, user interaction, varying operating system (iOS vs. Android) compatibility, etc. as well. As unit testing has commonly been integrated into application functionality, testing for the UI must be integrated as well. In this research project, I conducted an empirical study on two relevant UI testing tools, Xamarin.UITest and Exerciser Monkey, on a cross-platform application developed with Xamarin Forms, a cross-platform framework.

Problem Statement

In 2020, studies show that there were 272.6 million smartphone users in the United States [1]. In addition to the immense volume of smartphone devices, there is also a wide plethora of applications available for download. In 2020, studies show that there were 5.06 million active apps on the app store [2]. With the use of mobile applications skyrocketing, in order to stay at the top of the market, app developers must be extra careful to not cause any bugs, both function-wise and UI-wise in any of their user's varying phone-model use. Applications that have less-than-perfect UI's tend to be regarded as unprofessional or even sketchy, and often lead to that application being removed from their devices.

Motivation

Cross-platform development is a highly relevant process in mobile application development. For this reason, I am currently developing my own cross-platform mobile application which has a heavy focus on user experience, and the topic of UI testing has become extremely relevant to me. My selected framework for cross-platform compatible development, Xamarin Forms, comes with its own testing framework for creating automated user interface tests, Xamarin.UITest [3][4]. However, this testing framework does not auto-generate tests itself. Therefore, due to its lack of auto-generation, I have decided to not only explore the benefits of Xamarin.UITest, but also compare it with the popular test-generating monkey tool for mobile UI testing – Exerciser Monkey [5]. Exerciser Monkey generates pseudo-random streams of user events to the application of choice and outputs the results in a single file. Throughout this study, I compare the efforts and results of Xamarin.UITest with the monkey tool. My goal is to see if the effort

involved with writing the tests yields a more beneficial result than the Exerciser Monkey tool. I will base my comparisons on the simplicity of use, time spent setting up, and clarity of the results.

Limitations of Existing Work

Although Xamarin Forms has its own testing framework with many benefits such as the App Center Test, this framework only allows for manually-written tests. There is no ability to have tests automatically generated. Though not a limitation, Xamarin.UITest does require the developer to prep the source code before starting testing. Additional identifiers must be added to each UI component that will be included in the testing. Further, depending on the code coverage the developer wants to include, it can become quite time-consuming to write the individual tests.

Upon my research, I found the best tool for automatically generating tests for UI testing to be Exerciser Monkey. This tool is only compatible with Android applications, and cannot test the UI for applications running on iOS. Further, the monkey tool can only be used on the current device running the application. For the purpose of testing, if the UI remains consistent on various devices with varying screen size and resolution, the best use of the Exerciser Monkey would be to run it on every device on the market. However, this is an incredibly impractical way to test UI bugs.

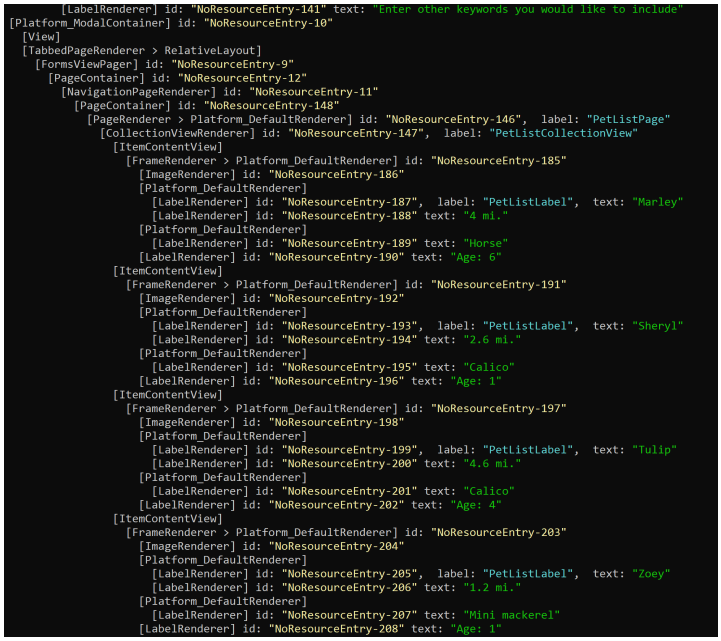
Background

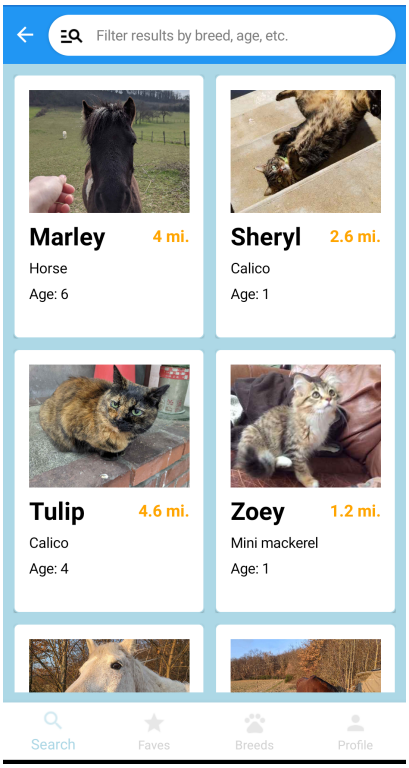
Xamarin Developing and Testing Frameworks

Xamarin is an open-source app platform that offers cross-platform compatibility. There is a framework called Xamarin.Forms that allows developers to build native UIs for iOS, Android, and Windows applications from a single, shared C# codebase. This ability significantly increases the speed of app development. With this practicality, Xamarin.Forms have become a popular choice for mobile app development and has been used by well-known companies such as Alaska Airlines [6]. There also exists a native Xamarin which simply focuses on one OS at a time. Xamarin.UITest allows testing both apps written with Xamarin.iOS and Xamarin.Android, allowing developers to test both targeted systems at the same time.

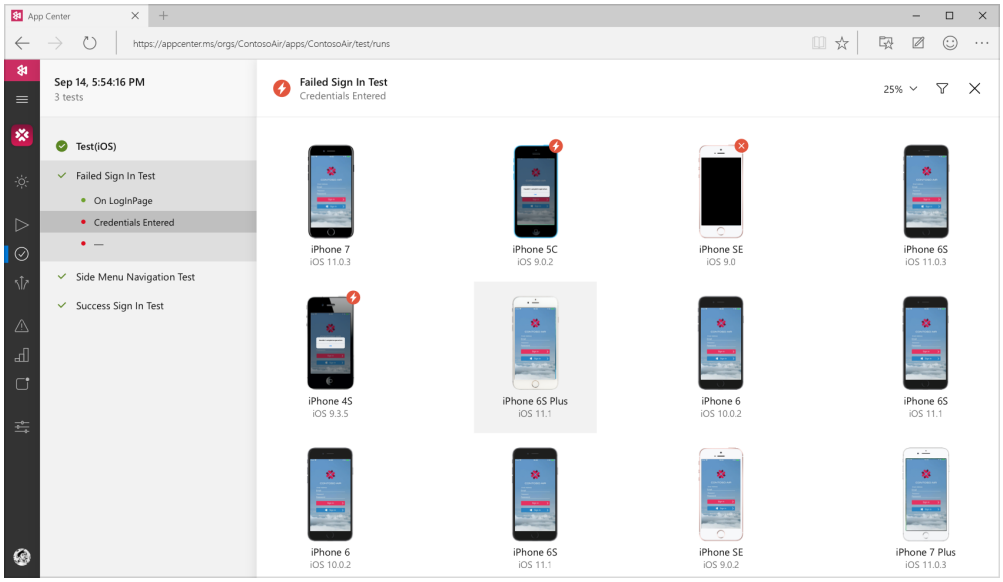
Xamarin.UITest functions by automating the user interface by activating controls on the screen and providing input anywhere a user would normally interact with the application. Developers must first assign special identifiers called AutomationId’s to the UI components to enable tests

to press a button or enter text in a box. When writing tests, it's also possible to specify categories to the tests in order to group tests together to create test suites. Xamarin.UITest also comes with a tool called REPL, which allows the developer to explore the application's UI [7]. Using the tree command within REPL prints out all the UI components in a hierarchical view present on the currently displayed application screen. The below screenshots show the REPL tree view of the screen shown side by side.





An additional tool associated with Xamarin.UITest is the App Center Test, formally known as the Xamarin Test Cloud. The App Center Test is a test automation service for mobile apps which allows developers to run their test cases on hundreds of unique device models and operating systems that are hosted in Microsoft's data center [8]. Generated assets, such as screenshots and device logs, are kept for test reports.



Exerciser Monkey Tool

Exerciser Monkey is an external tool that follows the same concepts as a traditional monkey testing tool. Based on python, it is a testing script process that can be run against installed applications using the command line with optional various command options such as *--throttle 500* and *--pct-touch 20*. It is important to note that the Exerciser Monkey tool only works for Android OS, and is also available as a plugin for Android Studio. When run, this tool randomly inputs user events such as clicks, touches, or gestures, as well as several system-level vents. Once the testing process is done, a log file of the events and crashes (if any) is generated. For this reason, the following project will focus on the Android version of my mobile application.

Approach

Overview

For the purpose of this study, I compare the efforts and results of Xamarin.UITest with the monkey tool. Though Xamarin.UITest requires that I write the tests myself, I would like to see if the effort involved with writing the tests yields a more beneficial result than the Exerciser Monkey tool. I will base my comparisons on the simplicity of use, effort and time spent setting up, and clarity of the results. I will then offer my discussion on how the tools could be improved.

Implementation

Xamarin.UITest

I first set up the environment for using the Xamarin.UITest tool. I added the AutomationId's to all UI components that I wanted to involve in the tests. I then had to export the APK of my mobile application to be able to reference it in the testing framework. Finally, I wrote multiple test cases highlighting different UI features of my application. The ValidZipEntered() test first identifies the EnterLocation component ID, which represents an entry box, and enters a valid zip code such as 45140 into the text field. Following this entry, the test then taps on the FindPetsButton component ID, which represents a button that should lead the user to the main page of the application. The app.Repl() command allows me to explore the UI hierarchy of the destination page in the terminal; this does not actively play a role in the test case. Finally, the test case asserts that the MainPageLabel component ID is identifiable on the active screen.

```
[Test]
0 references
public void ValidZipEntered()
{
    Query mainPageLabelQuery = x => x.Marked("MainPageLabel");

    var inputToValidate = "45140";
    app.EnterText("EnterLocation", inputToValidate);
    app.Tap("FindPetsButton");

    app.Repl();

    Assert.IsTrue(app.Query(mainPageLabelQuery).Any());
}
```

Please refer to the presentation for a demo of a test case written with Xamarin.UITest.

Exerciser Monkey

The Exerciser Monkey tool did not require its own setup for usage, however, it is necessary to have the application installed on the connected device or running on the emulator. The package name of the application must also be identified to include in the command line command for the tool. For the below example, the monkey command begins with *adb shell monkey*, next followed by *-p com.companyname.petsinsights_all* with -p representing 'package' and the following string being the name of my application package. Next, the *-v 3000* signifies that I want to run 3000 events on the application, *--ignore-crashes* is added to continue running the tool even if one of the 3000 events leads to a crash. Then, the *--pct-nav 50* signals that I only want 50 percent of the events to be navigation events (such as swiping up, down, left, right). and *--pct-touch 20* signals that I want 20 percent of the events to be touch events. Finally, I store

the results of the tests into the petinsightsapp_log.txt file.

```
C:\Users\Mailys Surface Pro>adb shell monkey -p com.companyname.petinsights_all
 -v 3000  --ignore-crashes --pct-nav 50 --pct-touch 20> petinsightsapp_log.txt
```

During testing, I tried different variations of the command options to see what results I could get from them. Please refer to the presentation for a demo of the Exerciser Monkey tool being run against my application.

Results

Effort and Time Setting Up

The Xamarin.UITest tool lacked a clear user manual and mostly had to piece together its usage via sample repositories who incorporated the framework in their own application. Besides that, the setup and test-writing took about a total of two hours. Meanwhile, the Exerciser Monkey tool required no setup, and minimal effort was needed to extract the package name of the application I wanted to test. This tool is very straightforward.

Simplicity of Use

The Xamarin.UITest test scripts are written in a highly human-readable way and made it very simple to write and understand the tests. This tool also utilizes a very clean user interface once the tests have been written to distinguish if the test has been run and if it passed or failed. If the test failed, there is a clear explanation detailing why it failed. As for the Exerciser Monkey tool, this tool was used only through the command-line interface and only required a single command to run all the tests.

Clarity of Results

The Xamarin.UITest framework had a very organized layout for test status and results. It indicates if a test has not been run yet, and includes a green checkmark if the test passed, or a red x mark if the test failed. Furthermore, if the test failed there is a detailed explanation included. In the monkey tool, generated tests and results or crash reports are printed out to a log report after all tests have been run. Depending on the number of tests run, this log report can get very busy and is difficult to read through. As we can see below, it is clear that this particular test that was run included trackball movements and touch events, but simply from reading this we are unable to directly understand where the events happened without investigating the screen coordinates.

```
#Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10200000;
component=com.companyname.petinsights_all/crc64981426c216e4e19e.SplashActivity;end
    // Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
    cmp=com.companyname.petinsights_all/crc64981426c216e4e19e.SplashActivity } in package
    com.companyname.petinsights_all
:Sending Trackball (ACTION_MOVE): 0:(3.0,2.0)
:Sending Touch (ACTION_DOWN): 0:(789.0,472.0)
:Sending Touch (ACTION_UP): 0:(795.13525,477.0497)
:Sending Touch (ACTION_DOWN): 0:(674.0,2043.0)
:Sending Touch (ACTION_UP): 0:(686.0367,2043.3145)
:Sending Touch (ACTION_DOWN): 0:(922.0,1753.0)
:Sending Touch (ACTION_UP): 0:(927.9895,1754.7134)
:Sending Touch (ACTION_DOWN): 0:(1076.0,2072.0)
:Sending Touch (ACTION_UP): 0:(1077.166,2070.9062)
:Sending Touch (ACTION_DOWN): 0:(493.0,1261.0)
:Sending Touch (ACTION_UP): 0:(480.65967,1246.4891)
:Sending Touch (ACTION_DOWN): 0:(370.0,1538.0)
:Sending Touch (ACTION_UP): 0:(383.91525,1530.2421)
:Sending Trackball (ACTION_MOVE): 0:(1.0,-2.0)
    //[calendar_time:2021-04-19 16:54:43.085  system_uptime:311888955]
    // Sending event #100
```

Analysis of the results

The Xamarin.UITest framework helped for checking UI bugs in other devices to guarantee that the application's UI remained consistent in all devices, screen sizes, and resolutions. However, it's important to note that it may be less convenient to add the Xamarin.UITest functions after development have been completed as this would require digging through all the UI components within the code, but I believe it is still worth the effort to incorporate the tests later.
The Exerciser Monkey tool helped identify unexpected behavior and guided me to implement more checks in my code. Overall, it is a quick tool to test a lot of various behaviors and catch unexpected crashes.

Insights

Some important lessons I learned through this experience are to build the tests as you are developing your functions and designing your pages. Adding the tests afterward requires more digging, but is possible and may be worth it if your application is to be used on a wide array of devices. And the existing monkey tool is great for investigating unexpected crashes, but tracing every test conducted by the tool is a waste of time due to its difficult-to-interpret nature.

Limitations

The Monkey tool is only applicable to Android applications, and I have been unable to find a comparable tool for iOS applications. Meanwhile, Xamarin.UITest works for all devices no matter the operating system (also works for web applications). My goal was to test the UI of my cross-platform compatible application, but I was unable to have the benefits of random testing with various devices and operating systems.

Discussion

Overall, this experience was personally beneficial to me as I was able to test aspects of my project app. Even though it's less convenient to add the Xamarin.UITest functions after development, I personally believe it's still worth the effort to incorporate these tests in.

Future Work

My next step in this study would be to create a user interface for the Exerciser Monkey tool. I think it would be very beneficial to have a more organized view of the tests that were generated and run, along with a quick view of which tests passed and which tests failed. I would be curious to investigate how to improve the aspect where only the coordinates of the events are reported, to try to associate these coordinates to actual UI components in the application pages.

References

1. https://www.statista.com/statistics/201182/forecast-of-smartphone-users-in-the-us/#:~:text=For%202020%2C%20the%20number%20of,estimated%20to%20reach%20294.15%20million.
2. https://sensortower.com/blog/app-store-growth-forecast-2020#:~:text=According%20to%20our%20projections%2C%20the,the%20end%20of%20this%20year.
3. https://dotnet.microsoft.com/apps/xamarin/xamarin-forms
4. https://docs.microsoft.com/en-us/appcenter/test-cloud/frameworks/uitest/
5. https://developer.android.com/studio/test/monkey
6. https://azure.microsoft.com/en-us/resources/videos/alaska-airlines-visual-studio-team-services-xamarin/
7. https://docs.microsoft.com/en-us/appcenter/test-cloud/frameworks/uitest/features/repl?tabs=vswin
8. https://docs.microsoft.com/en-us/appcenter/test-cloud/