UI Automation Test Guide

1. Introduction

Testing UI functionalities for apps will be very tedious if we human beings have to touch the screen every time in order to see whether UI works or not. Therefore, it would save a lot of work if we can make some scripts or code that can directly test UI. Fortunately, we can use adb, which stands for Android Debug Bridge, to have us automate our test. This guide mainly tells how to simulate some simple user interactions, such as tap, swipe, and typing characters by using adb. Next, we are going show you how to set up a series of actions for UI testing, which is the main component of our guide.

2. Set Up ADB

We won't talk a lot about how to set up adb environment since there is already an official link to help you set up adb and there are a lot of resources on the internet, simply Google it will be enough if you encounter any problem during set up.

3. Introduction to ADB

Android Debug Bridge is a very powerful tool for developers to dive deep into the Android system, but we won't talk more about it since it is irrelevant and useless to this guide. Google it if you are interested in it because I did when I first used adb.

Let us first talk about how to find your adb!

1. Simply open your terminal (if you are a Mac user, otherwise figure out a way to do it over Google)
2. Type command: `cd /Users/(your user name)/Library/Android/sdk/platform-tools`
3. Type command: `ls`
4. Then you will find out that among all the listed file, there is an executable file called "adb", there you go, you find it!

4. Find your Android device

1. Plug in your Android device with USB
2. Type command: `./adb devices`
3. Then you will see a list of Android devices
4. If you want to execute some command on a certain android device, just type command: `./adb –s` device-id command-you-want-to-execute

Right now, we have successfully connected our Android device with our computer! After Introduction to adb input command, we will soon be able to try out our own adb UI test!

5. Introduction to ADB Input Command

The input command is used to simulate user inputs, you can perform several kinds of user input behavior in order to fully test your User Interface. First let's get into the shell command mode of the Android device by simply typing the command: `./adb shell`

Now you should be get into the android device command line, it's just like SSH to remote server! You type whatever Linux command you want (but not traceroute as far as I know) since Android system is Linux based.

Next, let us type in command: `input`

The output of the command will be a brief list of what options the command has. In order to perform our UI test, it would be enough for us by using 4 out of 6 options from “The commands and default sources are:”. In our guide, we will only cover text, keyevent, tap and swipe since the rest of two options is kind of useless.

6. **input text “string”**

   Be careful with this command! If the typing keyboard window is not open, the order of some characters will be messed up. Make sure you tap on the text area first!
   Since we haven’t talked about tap command, let us first use our actual finger to tap on any text area you like, as long as the keyboard shows up.
   Next, in your terminal, type command: input text "You are beautiful" (Don’t simply just copy paste since the double quote character in MS Word is different from typing the double quote characters in terminal!)
   After that, you have finished your first action on Android device! Isn’t it amazing that you can simulate human activities via another machine!

7. **input keyevent event-code**

   After typing in the string “You are beautiful” into the text area, you may want to either create a new line or hit enter button to ask Google search engine to search what you have typed, what would you do? Since \n character does not work in the case! Here, another option for input, which is keyevent is coming in to play an important role.
   There is a table that match each number to a certain key stroke on the keyboard:

```
0 --> "KEYCODE_UNKNOWN"
1 --> "KEYCODE_MENU"
2 --> "KEYCODE_SOFT_RIGHT"
3 --> "KEYCODE_HOME"
4 --> "KEYCODE_BACK"
5 --> "KEYCODE_CALL"
6 --> "KEYCODE_ENDCALL"
7 --> "KEYCODE_0"
8 --> "KEYCODE_1"
9 --> "KEYCODE_2"
10 --> "KEYCODE_3"
11 --> "KEYCODE_4"
12 --> "KEYCODE_5"
13 --> "KEYCODE_6"
14 --> "KEYCODE_7"
15 --> "KEYCODE_8"
16 --> "KEYCODE_9"
17 --> "KEYCODE_STAR"
18 --> "KEYCODE_POUND"
19 --> "KEYCODE_DPAD_UP"
20 --> "KEYCODE_DPAD_DOWN"
21 --> "KEYCODE_DPAD_LEFT"
22 --> "KEYCODE_DPAD_RIGHT"
23 --> "KEYCODE_DPAD_CENTER"
24 --> "KEYCODE_VOLUME_UP"
25 --> "KEYCODE_VOLUME_DOWN"
26 --> "KEYCODE_POWER"
27 --> "KEYCODE_CAMERA"
```

```
28 --> "KEYCODE_CLEAR"
29 --> "KEYCODE_A"
30 --> "KEYCODE_B"
31 --> "KEYCODE_C"
32 --> "KEYCODE_D"
33 --> "KEYCODE_E"
34 --> "KEYCODE_F"
35 --> "KEYCODE_G"
36 --> "KEYCODE_H"
37 --> "KEYCODE_I"
38 --> "KEYCODE_J"
39 --> "KEYCODE_K"
40 --> "KEYCODE_L"
41 --> "KEYCODE_M"
42 --> "KEYCODE_N"
43 --> "KEYCODE_O"
44 --> "KEYCODE_P"
45 --> "KEYCODE_Q"
46 --> "KEYCODE_R"
47 --> "KEYCODE_S"
48 --> "KEYCODE_T"
49 --> "KEYCODE_U"
50 --> "KEYCODE_V"
51 --> "KEYCODE_W"
52 --> "KEYCODE_X"
53 --> "KEYCODE_Y"
54 --> "KEYCODE_Z"
55 --> "KEYCODE_COMMA"
56 --> "KEYCODE_PERIOD"
57 --> "KEYCODE_ALT_LEFT"
58 --> "KEYCODE_ALT_RIGHT"
59 --> "KEYCODE_SHIFT_LEFT"
60 --> "KEYCODE_SHIFT_RIGHT"
61 --> "KEYCODE_TAB"
62 --> "KEYCODE_SPACE"
63 --> "KEYCODE_SYM"
64 --> "KEYCODE_EXPLORER"
65 --> "KEYCODE_ENVELOPE"
66 --> "KEYCODE_ENTER"
67 --> "KEYCODE_DEL"
68 --> "KEYCODE_GRAVE"
69 --> "KEYCODE_MINUS"
70 --> "KEYCODE_EQUALS"
71 --> "KEYCODE_LEFT_BRACKET"
72 --> "KEYCODE_RIGHT_BRACKET"
73 --> "KEYCODE_BACKSLASH"
74 --> "KEYCODE_SEMICOLON"
75 --> "KEYCODE_APOSTROPHE"
76 --> "KEYCODE_SLASH"
77 --> "KEYCODE_AT"
78 --> "KEYCODE_NUM"
79 --> "KEYCODE_HEADSETHOOK"
80 --> "KEYCODE_FOCUS"
81 --> "KEYCODE_PLUS"
82 --> "KEYCODE_MENU"
83 --> "KEYCODE_NOTIFICATION"
```

```
84 -->  "KEYCODE_SEARCH"
85 -->  "TAG_LAST_KEYCODE"
```

To simulate the keyboard behaviors you want, simply follow this table. You can type in string with a combination of keyevent but it is not necessary since we already have input text command! But we can type “Enter” by using the key code for key event! The following is an example:

If we want to type “Enter” after inputting “You are beautiful” in Google search bar, we can simply type command: `input keyevent 66`

Then your input will be processed by Google search engine!

8. **input tap x y**

Tapping is easy! Just type command: input tap x y

That’s it.

But there is one thing that you should notice about, and that is x is horizontal distance from top left corner and y is vertical distance from top left corner. For our testing cell phone, which is Alcatel Onetouch, 0 <= x <= 1024, 0 <= y <= 2048

So right now go back to your home page, if you type the command:

`input tap 500 1700`

You will see your menu button has been tapped and the menu is shown up

9. **input swipe x1 y1 x2 y2 (duration)**

Swipe also plays a huge part in user interaction with UI, so it is very important to make sure that our swipe simulation works exactly the same as we expected. This command is a little bit complicated. For swiping, we must have a starting coordinate and an ending coordinate, just like a vector in mathematics. Therefore, x1 y1 is the starting point of the swiping and x2 y2 is the ending point. Notice that there is another argument, which is duration, is the time it takes to complete swiping on screen with unit of microsecond.

Let us try these two commands on your home page:

```
input swipe 0 0 0 1500 6000
input swipe 0 0 0 1500
```

These two commands do the same thing except it takes the first command 6 seconds to complete the whole task.

10. **Series of Testing Actions**

The above four sections are talking about how to test your UI by just a single command. Now, let’s start testing our device by a series of actions consist of texting, tapping, keyevent and swiping. The following example shows you how to create your test cases on your local machine and execute the test case to test your Android UI via adb.

1. In terminal, create your own testing file in your testing directory and type command: `touch test.sh`
2. Change the mode of test.sh to executable by: `chmod u+x test.sh`
3. Open the file via text editor (Sublime, vim, emacs, nano, etc.) and type `#!/bin/bash` on the first line

4. Since shell command has limited buffer length, it is not a good idea to put in hundreds of actions into a single adb command. I would recommend to group some small set of actions into a single command.
   Now let's try this by typing the command:

   ```
   ./Users/(your user name)/Library/Android/sdk/platform-tools /adb shell "input touchscreen swipe 0 500 500 500;input touchscreen swipe 500 500 0 500"
   ```

   The above command is a set of actions trying to swipe on the screen from 0, 500 to 500, 500 and swipe back. Let's save the file and execute the bash script and we can observe what we have expected. After saving the file, type command:
   `./testFile`
   then you will see your input is being executed by Android device.

For an even more complicated set of actions, try enter the following commands into the file:

```
./Users/(your user name)/Library/Android/sdk/platform-tools /adb shell "input touchscreen swipe 0 500 500 500; input touchscreen swipe 500 500 0 500"

./Users/(your user name)/Library/Android/sdk/platform-tools /adb shell "input touchscreen swipe 500 0 500 500; input touchscreen swipe 500 1700 500 0"

./Users/(your user name)/Library/Android/sdk/platform-tools /adb shell "input touchscreen tap 500 300; input text 'hack your motherhackers'; input keyevent 66"
```

What the above commands do is: swipe right, swipe, left, tap Google search bar, input text "hack your motherhackers" and hit "enter" to search it.

Another way to create tests for a series of actions is to create the test cases inside the Android device. We need to create our test file in a different way and transfer that file to our Android devices, then use adb command to execute our input test files. We will still use the above example to demonstrate how to do it this way.

1. Create the test file as we did in the previous example
2. Since we will execute input commands directly from Android device, we do not need to execute adb every time, hence the content of the file looks like this:

```
input touchscreen swipe 0 500 500 500; input touchscreen swipe 500 500 0 500

input touchscreen swipe 500 0 500 500; input touchscreen swipe 500 1700 500 0

input touchscreen tap 500 300; input text 'hack your motherhackers'; input keyevent 66
```

3. Save the file and type the following command to transfer the file to Android device:
   `./Users/(your user name)/Library/Android/sdk/platform-tools/adb push testFile.sh /sdcard/`

4. Getting into the Android device in the corresponding directory and execute our test file
   `./Users/(your user name)/Library/Android/sdk/platform-tools/adb shell`

   `cd /sdcard/`

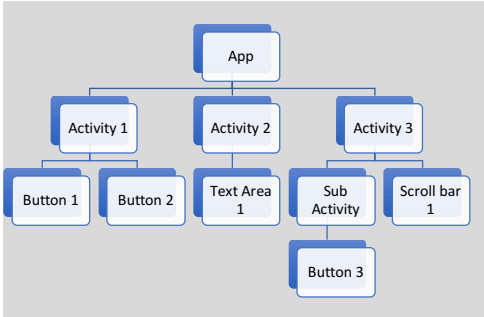
   `sh testFile`

11. Automate Testing Logistics

Hierarchy:

To fully test our application's User Interface, we need to be clear about what we need to have a clear picture of the structure of our apps, such as the hierarchy picture that is shown below. Based on application's relationship, we can have 3 levels of hierarchy for testing logistics, which are app level, activity and sub activity level and component level. There might be more levels, but as far as for UI testing's concern, reaching to the depth of component level is enough since we only care about the correctness of UI behaviors when we are testing all the possible user inputs and almost all user interactions are interacting at component level and activity level. Once we have a clear picture of app's hierarchy, we then can start planning and drafting our expected test results and how to create our test cases in an efficient way.

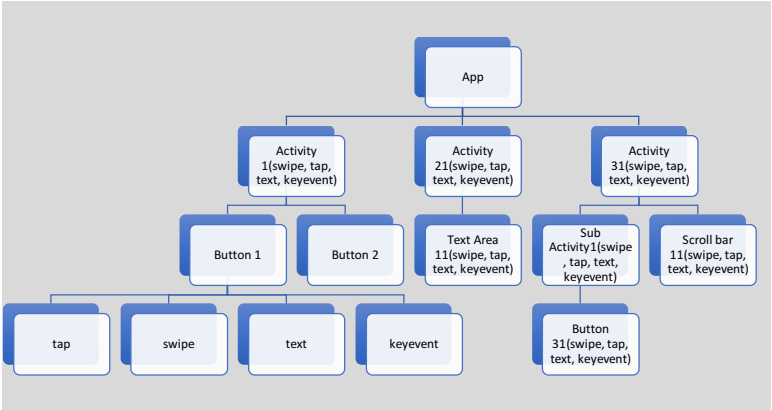
Testing hierarchy:



Expected Behavior and Result:

Expected Result is the very first thing we need to determine before testing on our UI because it is a standard to see whether the design and work flow is correct or not. Users mainly have four ways to interact with UI, which are tap, swipe, text and eventkey. Therefore, for each level component, it is necessary to know the expected behavior before testing.

Testing hierarchy with expected result level:



Test Case Unit:

During the UI test, we may have to click a certain button many times for different test cases. Therefore, creating reusable test cases is very important for clearness and efficiency. For example, “Button A” is being tested by an earlier single test case to just make sure the button works, but later on, there are a lot of other cases also involve clicking “Button A”, you definitely would prefer calling the previous “Button A” test case to perform the clicking instead of writing the whole input tap command again. It is not only more convenient but also more clear for testers to understand what this testing step is doing.