Unity Application Testing Automation with Appium and Image Recognition

Maxim Mozgovoy and Evgeny Pyshkin

Tsuruga, Ikki-Machi, Aizu-Wakamatsu,
Fukushima, 965-8580, Japan
{mozgovoy,pyshe}@u-aizu.ac.jp
http://www.u-aizu.ac.jp

Abstract. This work is dedicated to the problem of integrating simple functional tests (smoke tests) into the automated continuous integration pipeline. While functional testing is typically performed manually by the QA staff members, there is a large number of scenarios that can be automated, and readily available instruments, such as Appium and Calabash, designed for this task. Automated smoke testing is especially challenging for the applications with nonstandard GUI, such as games made with Unity. The only viable option in this case is to analyze screen content as a flat image and identify GUI elements with pattern matching algorithms. This approach is not novel, but its practical applicability and limitations are rarely discussed. We consider a case study of a mobile tennis game project, developed in Unity and covered with a suite of Appium-supported functional tests. We show how image matching capabilities of OpenCV library can be used in Appium tests to build a reliable automated QA pipeline.

Keywords: GUI; testing; computer game; automation; non-native; smoke test; OpenCV; Unity; Appium

1 Introduction

It is widely recognized that software quality assurance (QA) techniques appear at nearly every stage of software lifecycle, beginning from discovering requirements up to product deployment and maintenance. Specifically, professional software development methodologies emphasize importance of testing as a major dynamic software QA method. Particular attention is paid to testing automation which is an integral part of continuous integration pipeline – a process of daily automated build and deployment recommended by many experts for practical everyday use [10]. Automated tests became a core of certain practical approaches such as test-driven development (TDD) [7] or behavior-driven development (BDD) [18]. Simple automated tests are used for basic program elements, such as individual class methods or separate functions. Nevertheless, detailed functional testing (aimed to reveal whether a software meets requirement specifications) is still a complicated (and partially manual) process, typically performed by the QA staff.

However, certain relatively simple cases, known as smoke tests [15], can be automated. Smoke tests are aimed at performing some basic checkups: whether the program runs at all, is it able to open required windows, does it react properly to user inputs, etc. Automated user interface (UI) smoke tests should be able to access applications in the same way as users do, so they need to manipulate application's user interface. Specifically, testing graphical UI (GUI) provides interesting and nontrivial case of testing automation.

An important aspect of testing automation complexity (which is particularly significant for GUI testing) is a fragile test problem pointed by Meszaros as far back as 2007 [16]. A fragile test is a test which does not compile or does not work despite the fact that the code modification is not within the test scope. One of possible effects of such a fragility is test sensitivity to the changes in the application interface (especially in a case of GUI). The latter is one of reasons that application functional logic should not be tested via application's user interface, despite the fact that this rule is often violated in real life.

2 Instruments for GUI Testing

Existing tools for testing automation provide features for testing GUI applications in regular cases. For example, Jemmy library [4] allows to access programmatically many GUI elements: we can define and test situations conditioned by appearance of certain GUI components or user controls on the screen and by performing different operations such as pushing a button, selecting a menu item, scrolling a window, hovering an area, and so on.

Below there is an example of a typical testing scenario for an interactive computer game implemented in Java. Assume we have to test the initial preparations and assure that the game process goes to the end (in principle). First, we might have to set up the application frame window:

Defining actions executed before each unit test

```
@Before
public void setUp() {
  java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
      new GameFrame();
    }
  });
  // Attaching to the main frame
  mainFrame = new JFrameOperator();
  // Creating the object for event queue control
  mainQueue = new QueueTool();
  mainQueue.waitEmpty(200);
}
```

Then the *mainFrame* operator is attached to the application's main frame window, while *mainQueue* object is responsible for the operations with the event

queue. So, the next possible step is to define some testing scenario, for example, a game start process. The test sequence consists of the following operations:

1. Select the menu item *Game—>New;*
2. Select the level of players;
3. Push the button *"Start game"*;
4. Wait until the game is over (assuming that a game is over if its event queue remains empty for several seconds).

Defining a sample GUI test

```
@Test
public void testStartGame() {
    // Execute menu command
    JMenuBarOperator menuBar = new JMenuBarOperator(mainFrame);
    menuBar.pushMenuNoBlock(new String[] {"Game", "New" });
    JDialogOperator dialog = new JDialogOperator(
        "Select players");
    assertTrue(dialog.isVisible());
    mainQueue.waitEmpty(1000);
    // Select levels
    JRadioButtonOperator radio1 = new JRadioButtonOperator(
        dialog, "Beginner", 0);
    radio1.push();
    mainQueue.waitEmpty(1000);
    JRadioButtonOperator radio2 = new JRadioButtonOperator(
        dialog, "Master", 1);
    radio2.push();
    mainQueue.waitEmpty(1000);
    // Start the game
    JButtonOperator start = new JButtonOperator(dialog, "Start");
    start.push();
    // Wait for the end of the game
    mainQueue.waitEmpty(5000);
}
```

In process of test execution, we can see the windows appeared on the screen and actions processed during the game. The test is finished as soon as there are no more events in the event queue. We have to mention that the basic idea implemented in the frameworks similar to Jemmy is to separate testing of functional application logic from GUI testing. However, during testing of GUI applications we have to consider factors that might conduce to appearance of fragile or unstable tests.

On Windows one can also use Microsoft UI Automation [5]; similar solutions are also available on mobile platforms, e.g., Android UI Automator [2].

3 The Problem of Custom GUI Elements

The idea of hiding platform-specific UI automation frameworks behind a universal façade interface was recently implemented in tools such as Appium [1] and Calabash [3]. However, difficulties appear if an application does not rely on natively rendered GUI components of an underlying operating system and does not use standardized GUI libraries such as GTk, Qt, Swing, WinForms, etc. Such a situation is typical for computer games, where GUI elements are often hand-drawn and may change their onscreen positions. Furthermore, much of interaction is performed with non-GUI onscreen game objects, such as buildings, game characters, map elements, etc. As a result, a UI automation framework recognizes the main window of such an application as a plain graphical image containing no UI elements.

Similar problems might appear in non-game applications. One example is text recognition applications where texts are effectively available as images (this issue is specifically important for oriental languages [20]). Another case is checking a hand-drawn design model against an implementation (on web sites, on mobile applications running on a big variety of devices with different screens), and resolving layout and localisation issues (for example, the text strings might be out of pre-designed graphical area due to the particularities of a specific language). Dealing with GUI elements as with images might be required in electronic maps. Finally (and here we come to our particular case), there is a big domain of *Unity-based applications*.

In this contribution we discuss our experience in automating tests of a mobile game application with Appium using image recognition technologies. We use OpenCV library [8] in Appium test scripts to recognize game objects and hand-drawn Unity GUI elements in plain graphical data. We show that despite certain disadvantages this approach is feasible for a practical game project, and can be used to implement smoke testing as an element of a continuous integration pipeline in similar non-native GUI applications.

4 Unity GUI Testing with Appium

A developer of any automated GUI test suite needs to know whether the application under testing uses *native* GUI elements. Typically, a GUI is created with the help of numerous available frameworks; such frameworks are especially helpful for cross-platform development, since they often provide a universal multiplatform API. In turn, GUI frameworks normally rely on *API layering* or *GUI emulation* [9] (p. 5). The idea of API layering is to provide a universal wrapper for native GUI elements, provided by the underlying operating system. While such universal APIs usually implement only the most common GUI controls, found in all supported operating systems, they ensure native look and feel, which can be important or even required for business applications. An alternative approach, GUI emulation, relies on screen drawing functions to visualize GUI controls. This method ensures the same look and feel on each supported platform, and imposes no platform-specific restrictions on the set of available GUI elements.

From the perspective of GUI testing, there are subsequent important differences. Since the operating system manages native GUI controls, it "knows" all onscreen GUI elements, and can potentially provide functions to manipulate them. In contrast, an emulated GUI for an operating system is just a flat canvas, used to draw graphical primitives, such as lines and circles. Therefore, such non-native GUI in general cannot be manipulated with the API of the underlying operating system. Since Unity attempts to reproduce the same look and feel on each supported platform, Unity applications normally follow GUI emulation approach.

4.1 Basic Appium Setup

Appium is a test automation framework designed to assist functional testing of compiled native (iOS, Android or Windows), and hybrid applications [11]. Appium-powered scripts access applications nearly the same way as end users do: such scripts are able to press buttons, select check boxes or radio buttons, enter text strings into edit boxes, examine content of labels, and proceed with mouse clicking on arbitrary areas. Appium is implemented as a client-server system where a server exposes a REST API. Particularly, Appium is responsible for the following activities:

1. Receiving connections from a client;
2. Listening for commands;
3. Physical execution of those received commands from an application under testing on Appium servers (directly or using external USB-connected mobile devices in case of mobile applications);
4. Responding by using an HTTP response representing the result of the command execution.

Thus, remote Appium clients connect to the servers and run test scripts that send commands for execution. Due to the supporting client libraries, test scripts can be authored in a variety of popular languages, including Java, Ruby, Python, PHP, JavaScript, and C#.

Native GUI elements of an application are accessed with a specialized API. For example, if there is a single textbox and a single OK button on the Android screen, one can simulate user input as follows:

User Input Simulation (Python)

```
e = appium.find_element_by_class_name
    ('android.widget.EditText')
e.send_keys("hello, world") ok =
    appium.find_element_by_class_name
    ('android.widget.Button')
ok.click()
```

Appium scripts can also take and examine screenshots of applications, which is a crucial ability in our case of custom Unity GUI elements.

4.2 Screenshot Analysis Using Image Recognition

At a glance it may seem that identifying objects of interest on the screen (such as GUI elements or game characters) can be reduced to the task of perfect matching of a bitmap image inside a screenshot. However, we can cite several factors confirming that such a naïve exact matching is insufficient:

- Onscreen objects may be rendered differently by different GPUs or due to different rendering quality settings;
- Since screens vary in dimensions, we need to scale pattern images, which causes distortions;
- Onscreen objects often intersect with each other. Therefore, approximate matching is necessary.

The idea of using image matching in Appium is discussed in several tutorials [13, 12]. Typically, authors suggest employing OpenCV library for approximate matching. We rely on OpenCV function *matchTemplate()* called with the flag *TM_CCOEFF_NORMED*. This allows us to get image similarity coefficient in a range of $[0 \dots 1]$ in order be able to analyze testing results from the viewpoint of UI elements recognition quality.

Unfortunately, *matchTemplate()* function is unable to match scaled patterns. Since game may run on devices with different screen sizes, we scale the screenshots to match the dimensions of the original screen used to record graphical patterns.

4.3 Smoke Tests in a Continuous Integration Pipeline

Let us remind that smoke tests are usually more complicated compare to regular unit tests. At the same time, smoke tests are not detailed enough so as to provide an exhaustive coverage of possible usage scenarios. What is nice about automated smoke tests is that they are reproducible and they support regressive testing. Moreover, a test suite may be extended as soon as some specific defects are discovered as a result of other QA activities.

In our case, the continuous integration setup relies on a popular build server TeamCity [14]. Since smoke tests might be time consuming, we do run them asynchronously. The testing subsystem periodically polls TeamCity to detect new finished builds, and and runs all the tests for them. All results are combined into the HTML-based report, indicating test outcomes. It is also possible to examine a detailed report (with screenshots) of each testing session, which helps to identify found problems.

Figure 1 shows the general organization of the GUI smoke test automation process implemented in our approach. The tests are executed on real mobile devices, connected via USB cables to two computers, running Appium. The first computer is a Mac mini that runs tests on iPad 3 and iPad Air. The second computer is a Windows-based PC with five connected Android devices: Nexus 7, ASUS MeMO Pad 7, Kindle Fire 5th Gen, Doogee X5 Max Pro, and Xiaomi Redmi Note 3 Pro. We use Plugable USB 3.0 7-port hubs that ensure simultaneous data transfer and battery charging.

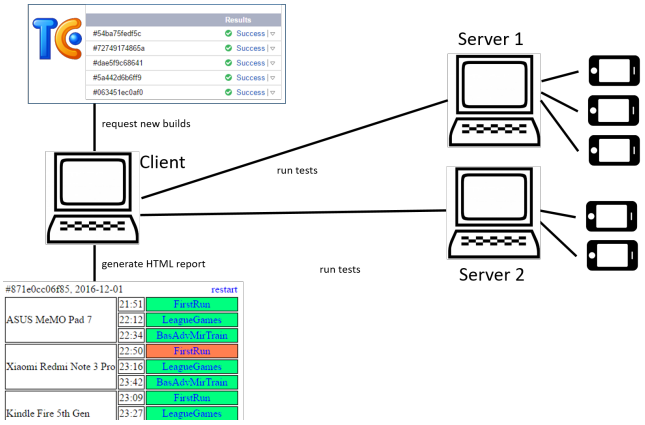


Fig. 1. Testing organization

5 Case Study: World of Tennis

We implemented automated smoke testing for the upcoming mobile tennis game *World of Tennis* [6], used in our previous experiments with believable and effective AI agents [19, 17]. Initially we planned to test only the most basic game functionality, such as creating a new user account and starting a training match. Later we managed to rely on automated GUI tests for a variety of needs:

- *First run and tutorial.* On the first run, the game performs the following actions: register a new online user account; show a tutorial; ask the user to test the system of character upgrades; play a training match. Therefore, the first game run requires several core subsystems to work properly. Successful first run is more than just a smoke test, it is a good indicator of a stable game build.
- *Online league games.* By playing several league matches, the testing system checks the stable work of a typical game routine.
- *Stress tests.* At night time, the system plays a large number (10–20) of league games to make sure the game is stable during prolonged (1.5–2.5 hour) game sessions.
- *Graphics and framerate control.* Since autotests produce detailed step-by-step reports with screenshots, it is convenient to use them to spot graphical glitches, texture distortions, and poor framerate (a framerate counter is displayed in each screenshot). Our experience shows that graphical glitches are often device-dependent, and framerate may drop unexpectedly in certain game situations due to behavior of a particular GPU, so automated tests can greatly assist, if not substitute, manual QA work scenarios.

Since game GUI elements are all hand-drawn, there is no option to rely on native UI automation capabilities. One may try to hardcode coordinates of UI

elements in the tests, but this method can only assist game control, while we also need to recognize game events and the presence of certain onscreen elements to react accordingly. Furthermore, different devices have different screen sizes and aspect ratios, which makes coordinate-based approach fragile.

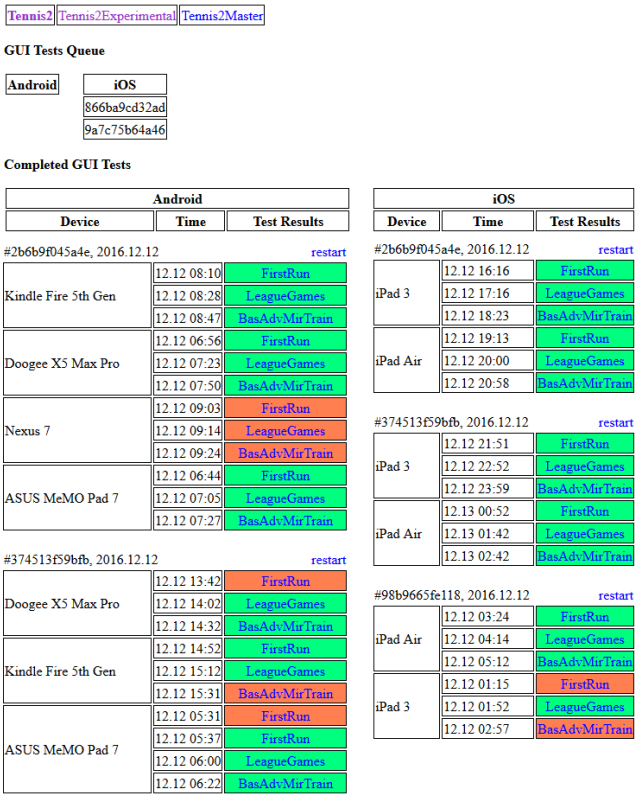
In general, all our tests follow the same routine: take a screenshot, detect the presence of certain GUI elements (see Figure 2), react accordingly, repeat. The largest performance bottleneck in this scenario is the process of acquiring the next screenshot, which might take up to several seconds depending on a device.

Fig. 2. Examples of UI elements and game characters

Our experiments show that the reliability of pattern matching in the scripts varies depending on a task. Static GUI elements (such as buttons or menus) can be almost always reliably recognized with high degree of similarity $(0.90 \ldots 0.98)$, according to OpenCV reports. Certain elements interfere with the background and thus yield lower ratios of $(0.63 \ldots 0.65)$ (currently we consider an image matched if the similarity ratio of its best match is 0.60 or higher). For example, the main menu icon (see Figure 2) is placed against the sky with moving clouds, making perfect template matching impossible. When variations of the same GUI elements cause mismatches, we keep a list of several common images, and match all of them before reporting the absence of the element on the screen.

The opposite case is *false positive*, when the system detects the presence of a certain element, actually not shown on the screen. Typically, a false positive is triggered when small similar-looking graphical elements are confused with each other. To prevent this situation, we try to match larger regions providing more context. For example, in *World of Tennis*, the Skip button is always placed next to a checkbox, so we match the whole button/checkbox region. If this method still does not work, we match all possible candidate elements, and report the element having the highest similarity ratio with its identified match.

Figure 3 demonstrates a test report generated for a certain stage of *World of Tennis* testing process performed for a selection of Android and iOS devices. In Figure 4 you can see a series of fragments from the detailed report for an example of *LeagueGames* test run for Kindle Fire 5th Gen Android device. In case of test failure the analysis of such a detailed report allows us to understand the reasons



Tennis2 | Tennis2Experimental | Tennis2Master

GUI Tests Queue

Android	iOS
	866ba9cd32ad
	9a7c75b64a46

Completed GUI Tests

Android

Device	Time	Test Results
#2b6b9f045a4e, 2016.12.12		restart
Kindle Fire 5th Gen	12.12 08:10	FirstRun
	12.12 08:28	LeagueGames
	12.12 08:47	BasAdvMirTrain
Doogee X5 Max Pro	12.12 06:56	FirstRun
	12.12 07:23	LeagueGames
	12.12 07:50	BasAdvMirTrain
Nexus 7	12.12 09:03	FirstRun
	12.12 09:14	LeagueGames
	12.12 09:24	BasAdvMirTrain
ASUS MeMO Pad 7	12.12 06:44	FirstRun
	12.12 07:05	LeagueGames
	12.12 07:27	BasAdvMirTrain
#374513f59bfb, 2016.12.12		restart
Doogee X5 Max Pro	12.12 13:42	FirstRun
	12.12 14:02	LeagueGames
	12.12 14:32	BasAdvMirTrain
Kindle Fire 5th Gen	12.12 14:52	FirstRun
	12.12 15:12	LeagueGames
	12.12 15:31	BasAdvMirTrain
ASUS MeMO Pad 7	12.12 05:31	FirstRun
	12.12 05:37	FirstRun
	12.12 06:00	LeagueGames
	12.12 06:22	BasAdvMirTrain

iOS

Device	Time	Test Results
#2b6b9f045a4e, 2016.12.12		restart
iPad 3	12.12 16:16	FirstRun
	12.12 17:16	LeagueGames
	12.12 18:23	BasAdvMirTrain
iPad Air	12.12 19:13	FirstRun
	12.12 20:00	LeagueGames
	12.12 20:58	BasAdvMirTrain
#374513f59bfb, 2016.12.12		restart
iPad 3	12.12 21:51	FirstRun
	12.12 22:52	LeagueGames
	12.12 23:59	BasAdvMirTrain
iPad Air	12.13 00:52	FirstRun
	12.13 01:42	LeagueGames
	12.13 02:42	BasAdvMirTrain
#98b9665fe118, 2016.12.12		restart
iPad Air	12.12 03:24	FirstRun
	12.12 04:14	LeagueGames
	12.12 05:12	BasAdvMirTrain
iPad 3	12.12 01:15	FirstRun
	12.12 01:52	LeagueGames
	12.12 02:57	BasAdvMirTrain

Fig. 3. Fragments from test run reports for a number of Android and iOS devices

of a discovered failure and makes possible to reconstruct the situation followed by this failure.

Our experience shows that most test failures are caused with unwanted changes between two consecutive actions. For example, the test script detects a certain onscreen element and tries to click it, but during this action the element happens to be covered by an unexpected pop-up modal dialog or by another game element. However, modal dialogs may disrupt any GUI testing script, so this problem is not specific for our approach.

6 Discussion

Our experiments show that the procedures for testing automation of the applications with non-native GUI require the combined use of several technologies including traditional automated unit tests, functional testing frameworks (Appium in our case) and image recognition (OpenCV).

Fig. 4. *LeagueGames* test: detailed report fragments with a number of screenshots saved during the game run

As we demonstrated in our *World of Tennis* case study, there could be recognition failures (false positive and false negative matches), however, most of them can be solved with reasonable efforts. Our difficulties were mostly caused with other factors, such as complex application logic or Appium / iOS / Android quirks. From the viewpoint of efficiency analysis, It should be noted that due to using graphical information, this approach requires massive amounts of data to be sent over the network, so it may not be applicable if Appium clients and servers communicate via slow channels.

Started with simple smoke tests, we realized that the designed system has more potential, and can help us in other scenarios, most notably stress testing. Appium tests can play dozens of tennis matches with no breaks, thus providing enough proof that a particular build is stable. We also found the resulting HTML reports to be helpful in quick performance and graphical sanity analysis, since the framerate value is shown on every screenshot, and graphical glitches are easy to spot.

Image recognition algorithms are not often examined within the scope of software testing, so we believe that the proposed approach providing a feasible solution for everyday automated smoke testing could be considered as a possible extension of a discourse connected to tools and methods for software analysis and verification automation.

References

1. Appium. project homepage, `http://appium.io`, accessed: Nov 1, 2016
2. Automating user interface tests, `https://developer.android.com/training/testing/ui-testing/index.html`, accessed: Nov 1, 2016
3. Calabash. project homepage, `http://calaba.sh`, accessed: Nov 1, 2016
4. Jemmy framework. project homepage, `https://jemmy.java.net/`, accessed: Nov 20, 2016
5. Microsoft ui automation, `https://msdn.microsoft.com/en-us/library/windows/desktop/ee684009.aspx`, accessed: Nov 1, 2016
6. World of tennis. project homepage, `http://worldoftennis.com/`, accessed: Nov 20, 2016
7. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
8. Bradski, G., Kaehler, A.: Learning OpenCV: Computer vision with the OpenCV library. " O'Reilly Media, Inc." (2008)
9. Dalheimer, M.: Programming with Qt: Writing Portable GUI Applications on Unix and Win32, 2nd Ed. O'Reilly Media (2002)
10. Duvall, P., Matyas, S., Glover, A.: Continuous integration: improving software quality and reducing risk (2007)
11. Hans, M.: Appium Essentials. PACKT (2015), `https://www.packtpub.com/application-development/appium-essentials/`
12. Helppi, V.V.: Using opencv and akaze for mobile app and game testing (January 2016), `http://bitbar.com/using-opencv-and-akaze-for-mobile-app-and-game-testing`, accessed: Nov 2, 2016

13. Kazmierczak, S.: Appium with image recognition (February 2016), https://medium.com/@SimonKaz/appium-with-image-recognition-17a92abaa23d\#.oez2f6hnh, accessed: Nov 2, 2016
14. Mahalingam, M.: Learning Continuous Integration with TeamCity. Packt Publishing Ltd (2014)
15. McConnell, S.: Daily build and smoke test. IEEE software 13(4), 144 (1996)
16. Meszaros, G.: xUnit test patterns: Refactoring test code. Pearson Education (2007)
17. Mozgovoy, M., Purgina, M., Umarov, I.: Believable self-learning ai for world of tennis. In: Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG 2016). pp. 247–253. IEEE, IEEE (Sep 2016)
18. North, D.: Behavior modification: The evolution of behavior-driven development. Better Software 8(3) (2006)
19. Umarov, I., Mozgovoy, M.: Creating believable and effective ai agents for games and simulations: Reviews and case study. Contemporary Advancements in Information Technology Development in Dynamic Environments pp. 33–57 (2014)
20. Wong, F., Chao, S., Chan, W.K., Li, Y.P.: Recognition of chinese character in snapshot translation system. In: Audio Language and Image Processing (ICALIP), 2010 International Conference on. pp. 821–825. IEEE (2010)