UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Jarno Tuovinen

MOBILE GAME TEST AUTOMATION WITH IMAGE RECOGNITION

Master's Thesis
Degree Programme in Computer Science and Engineering
Feb 2018

Tuovinen J. (2018) Mobile game test automation with image recognition. University of Oulu, Degree Programme in Computer Science and Engineering. Master's thesis, 50 p.

ABSTRACT

In the sequel of mobile game industry, testing is recognized as a key factor that can either boost or halt the game development. Indeed, the growth speed of mobile industry has led to decreasing quality in the apps and the employed testing tools. The wide spectrum of device hardware and software, so called fragmentation, makes it a lot slower to test the application on devices. Besides, there are more inherent technical issues to test games as compared to other mobile applications. The lack of efficient tools for testing robustness, quality and computational efficiency has been acknowledged by the game industry. This unmet need has motivated the work highlighted in this thesis which aims to explore easy and efficient approaches to generate test automation code for mobile games. First, to identify difficulties, challenges and good practices that mobile game developers face, a questionnaire-based analysis has been conducted and evaluated. Second, a new Android mobile game testing tool, called MAuto, has been developed and deployed. MAuto is developed to be compatible with Appium, open source test automation framework. MAuto is a tool that utilizes its user's input to generate Appium test automation code which can be run on the same or any other Android device. The system consists of three elements: the user, the browser, and the mobile device. Once the user has launched MAuto, all the interaction between the user and the tool happens with the browser. During the rerun, MAuto utilizes AKAZE algorithm to recognize the location of the object and repeats the user input with Appium. Consequently, MAuto has been used in developing test automation code to play the tutorial scene of the mobile game Clash of Clans. MAuto is designed to be a friendly mobile test creator that, with minimal amendments, can be exported to other testing applications in alternative platforms, e.g. iOS. Nevertheless, the thesis also acknowledges the limitations in the image-recognition based test automation methods where the negative impact of change of configuration, lack of luminosity, and non-exhaustive list of models cannot be ignored. On the other hand, the thesis also reviews the existing tools and current practices in mobile game testing, providing a comparable analysis for both researchers and developer communities.

Keywords: Test automation, Android, iOS, game testing, mobile game

Tuovinen J. (2018) Mobiilipelien automaatiotestaus kuvantunnistuksen avulla. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 50 s.

TIIVISTELMÄ

Mobiilipeliteollisuudessa testauksen on havaittu olevan avaintekijä, joka voi nopeuttaa tai hidastaa pelinkehitystä. Mobiiliteollisuuden kasvunopeus on johtanut applikaatioiden ja testaustyökalujen laadun heikkenemiseen. Laitteistojen ja ohjelmistojen laaja kirjo, niin kutsuttu fragmentaatio, hidastaa applikaatioiden testausta laitteilla. Lisäksi peleissä on enemmän niille ominaisia teknisiä ongelmia verrattuna muihin mobiiliapplikaatioihin. Peliteollisuudessa on huomattu, ettei ole olemassa riittävän tehokkaita työkaluja pelien vakauden, laadun ja tehokkuuden testaamiseen. Tämä diplomityö pyrkii vastaamaan tähän tarpeeseen. Työn tarkoitus on löytää helppoja ja tehokkaita lähestymistapoja testiautomaatiokoodin kehittämiseksi mobiilipeleille. Ensimmäiseksi on toteutettu kyselypohjainen analyysi, jotta voitaisiin selvittää mobiilikehittäjien kohtaamat vaikeudet ja haasteet sekä löytää heille parhaat käytänteet. Toiseksi on kehitetty uusi Android-pelien testaustyökalu, MAuto, joka on tarkoitettu erityisesti helpottamaan pelien testausta mutta joka toimii myös muilla applikaatioilla. MAuto on yhteensopiva Appiumin kanssa, joka on avoimen lähdekoodin testiautomaatioviitekehys. MAuto on työkalu, jolla voi luoda testaajan syötteistä Appium-testiautomaatiokoodia. Testikoodia voidaan ajaa uudelleen millä tahansa Android-laitteella. Järjestelmä koostuu kolmesta osasta: käyttäjästä, selaimesta ja mobiililaitteesta. Kun käyttäjä on käynnistänyt MAuton, interaktio käyttäjän ja työkalun välillä tapahtuu selaimessa. Kun testi ajetaan uudelleen, MAuto toistaa käyttäjän syötteen tunnistetussa sijainnissa hyödyntämällä AKAZE-algoritmia ja Appiumia. MAutoa on käytetty esimerkiksi luomaan testiautomaatiokoodi, joka pelaa Clash of Clans -peliä. MAuto on suunniteltu käyttäjäystävälliseksi, ja pienellä vaivalla sitä voidaan laajentaa kattamaan useampia käyttöjärjestelmiä, kuten iOS. Diplomityö käsittelee myös kuvantunnistukseen perustuvien työkalujen heikkouksia, kuten kuvantunnistuksen tunnistustehokkuutta verrattuna natiiviin objektintunnistukseen ja muuttuvien grafiikoiden aiheuttamia ongelmia testauksessa. Myös MAuton heikkouksia on käsitelty liittyen erityisesti sen nopeuteen ja toimintavarmuuteen peleissä, joissa grafiikan valaistus muuttuu tai peli on nopeatempoinen. Lisäksi työ arvioi olemassa olevia tekniikoita ja työkaluja ja tarjoaa vertailukelpoisen analyysin niin tutkijoille kuin kehittäjillekin.

Avainsanat: Automaatiotestaus, Android, iOS, pelitestaus, mobiilipeli

TABLE OF CONTENTS

FOREWORD

This thesis has been done with Bitbar Technologies. They already had a tool, Testdroid Recorder, which recorded user actions for Android applications and the actions could be replayed on the same or any other Android device. This tool didn't work with games and there wasn't any tool that could do it, so we got the idea to investigate it further.

I've been working on my thesis very long and I'd like to thank everybody who has helped me in any way. I'm sure I've tested their patience a lot and I've almost given up many times. Special thanks to my wife Sirpa who has supported me the most and without her support this thesis wouldn't have ever been finished. Also, Saija, Ismo, Anita and Kari have helped a lot not to forget supervisors Mourad, Kaj, Huber and Vassilis.

Oulu, Finland February 5, 2018

Jarno Tuovinen

ABBREVIATIONS

ADB	Android Debug Bridge
AKAZE features	Accelerated version of KAZE features
AOS	Additive Operator Splitting
API	Application programming interface
APK	Android application package
AUT	Application under test
CoC	Clash of Clans
CSS	Cascading Style Sheets
DUT	Device under test
E2E	End to end
FED	Fast Explicit Diffusion
GITR	GoogleInstrumentationTestRunner
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
iOS	Mobile operating system developed and distributed by Apple Inc.
JVM	Java virtual machine
KAZE features	2D feature detection and description method in a nonlinear scale space
KIF	Keep It Functional - Testing framework for iOS
M-LDB	Modified-Local Difference Binary
MMO/MMOG	Massive Multiplayer Online Game
OCR	Optical character recognition
OpenGL	Open Graphics Library
OS	Operating system
QA	Quality Assurance
QoS	Quality of Service
R&PB	Record and playback
R&R	Record and replay
SDK	Software Development Kit
SIFT	Scale Invariant Feature Transform
SURF	Speeded Up Robust Features
SUT	System under test
TDD	Test-Driven Development
UI	User Interface
VNC	Virtual Network Computing
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
XPath	XML Path Language

1. INTRODUCTION

The number of smartphones has increased rapidly in the last few years. This rapid adoption is in part due to the wide spectrum of applications that aid the users with their daily life activities. Even though games are not helping the users directly, they take 85% of all app revenue[1]. The growth speed of mobile industry has lead to decreased quality in the apps. Testing tools has not kept up with the growth and the quality assurance people has had hard time to make sure the apps work as expected. The testing tools provide better support for regular apps, but the game developers rely mainly on humans. Humans can test only limited amount of different kind of devices in given time as automated processes can test simple scenarios on more devices in less time.

However, current mobile application testing tools do not suit very well for mobile game testing. Games on mobile devices are harder to test than regular applications, because the objects in the games cannot be accessed as easily as objects in regular applications. The test instrumentation usually recognizes the visible objects by IDs, texts, descriptions and similar handles, but game content usually don't have these handles. Additionally, device diversity or fragmentation introduces extra challenges in the testing process as different devices have different operating system and specifications. As a result, the game is usually just tested in a minor range of devices using manual methods. To counter this problem, in this thesis, we propose a framework for testing, namely MAuto. The aim of the tool is to help the tester to create tests which work with Android games. The tests can then be used to rerun the tests on any other Android device.

MAuto can record tests from user interactions, export the tests to Appium tests for playback and everything is done with image recognition to avoid the problem of missing element handles. The focus is on mobile games, but it works with regular applications as well. To validate MAuto, tests are created with the tool for Clash of Clans[2] (version 8.551.4) from Supercell[3].

The rest of the thesis is organized as follows:

- *Chapter 2:* covers the essential terms and tools to understand what is mobile game testing with image recognition all about. This chapter digs deeper into test automation and game testing. Reader learns the basics of widely used image recognition algorithms. Background chapter also cover the related work and similar existing tools.
- *Chapter 3:* explains the core problem this thesis tries to solve.
- *Chapter 4:* reveals the questionnaire which was used to find the core problems in mobile game test automation.
- *Chapter 5:* describes the proposed testing framework called MAuto. This chapter explains how the problems were fixed and how the tool works.

[1]http://venturebeat.com/2016/02/10/mobile-games-hit-34-8b-in-2015-taking-85-of-all-app-revenues/

[2]https://play.google.com/store/apps/details?id=com.supercell.clashofclans

[3]http://supercell.com/

- *Chapter 6:* explains what metrics were used to measure and compare the tool to existing tools and methods. This chapter also contains the validation results.
- *Chapter 6:* discusses about the results. In this chapter we tell if the tool fulfills its purpose and are there any advantages for the user to use the tool.
- *Chapter 8:* concludes the thesis.

2. STATE OF THE ART

In this chapter, the essential terms and tools for this thesis are explained. Reader learns the basics of widely used image recognition algorithms and the basics of mobile test automation. This chapter will also explain related work.

2.1. Smartphones and mobile operating systems

In 2015 there were 341.5 million smartphone shipments in the world and the market grew 13.0% in one year. Samsung (21.4%) and Apple (13.9%) are the two biggest vendors by market share. Their combined share is 35.3% which leaves almost 65% on the table for other vendors. This means there are many vendors to share 341.4 million shipments, see Table 1.

Android and iOS are the two mostly used operating systems (OS) for smartphones. Android is developed by Google. The second quarter of 2015 Android had 82.8% and iOS had 13.9% market share[3]. This thesis will concentrate on Android, because at the moment it is dominating the mobile OS niche, see Table 2.

Table 1: Smartphone shipment market shares by IDC[2].

Period	Samsung	Apple	Huawei	Xiaomi	Lenovo[1]	Others
2015 Q2	21.4%	13.9%	8.7%	5.6%	4.7%	45.7%
2014 Q2	24.8%	11.6%	6.7%	4.6%	8.0%	44.3%
2013 Q2	31.9%	12.9%	4.3%	1.7%	5.7%	43.6%
2012 Q2	32.3%	16.6%	4.1%	1.0%	5.9%	40.2%

Table 2: Mobile OS market shares by IDC[3].

Period	Android	iOS	Windows Phone	Blackberry OS	Others
2015 Q2	82.8%	13.9%	2.6%	0.3%	0.4%
2014 Q2	84.8%	11.6%	2.5%	0.5%	0.7%
2013 Q2	79.8%	12.9%	3.4%	2.8%	1.2%
2012 Q2	69.3%	16.6%	3.1%	4.9%	6.1%

2.2. Fragmentation

Fragmentation refers to a concern over the number of distinct devices with different combinations of software and hardware. The number of distinct Android devices is increasing and there were 24093 distinct devices in August 2015[5]. In Android the

[1]Motorola is under Lenovo.

[2]http://www.idc.com/prodserv/smartphone-market-share.jsp

[3]http://www.idc.com/prodserv/smartphone-os-market-share.jsp

number of distinct devices is the biggest, but fragmentation is also a problem in other operating systems. For example Apple has 13 iPhone, 12 iPad and 7 iPod Touch models which makes 32 models total (August 2016). Many of the models have different hardware setups and combined with different OS versions the number of distinct devices will be hundreds[7][8]. Fragmentation can cause intermitted or "unrepeatable" bugs in applications[1]. Because of the large number of distinct devices it nearly impossible to test the application on every distinct device in real environment and to provide the best user experience the application must work flawlessly on any device.

In Table 3 are shown the yearly number of distinct Android devices by Open Signal[6]. The number seems to increase as Figure 1 shows.

Table 3: Disctinct devices by Open Signal[4][5].

Year	Disctinct devices
2012	3997
2013	11868
2014	18769
2015	24093

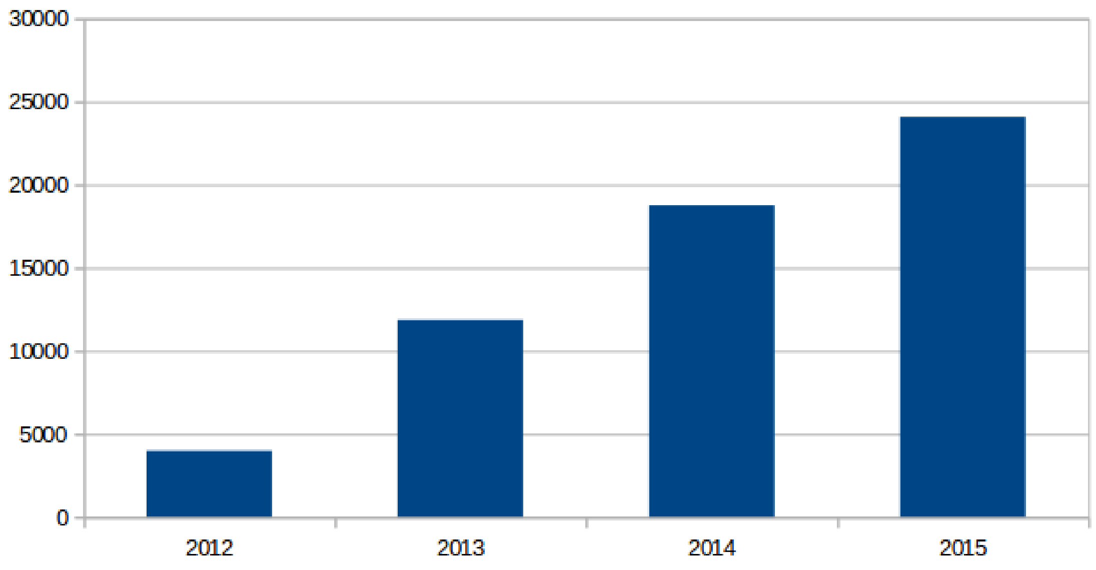


Figure 1: Table 3 presented in graphical format[4][5].

[4]http://opensignal.com/reports/fragmentation-2013/
[5]http://opensignal.com/reports/2015/08/android-fragmentation/
[6]http://opensignal.com/
[7]https://en.wikipedia.org/wiki/List_of_iOS_devices
[8]https://www.theiphonewiki.com/wiki/Models

2.3. Mobile device groups

Usually mobile devices are grouped into three groups: high-end, midrange and low-end devices. The boundaries between these three groups are not very clear[9]. The devices with the best processing power, display and latest operating system are in the high-end group. The low-end group contains the cheapest devices with not much processing power or any other hardware. In these devices the operating system tend to be very old as well. All the rest are grouped into midrange group. The boundaries can vary greatly between projects and requirements[2].

2.4. Software development for mobile devices

Mobile devices have some additional problems compared to traditional software development. The devices can provide much more value than desktop machines, because they have a lot of sensors for detecting movement, light, location, temperature, etc. The memory, battery power, screen size and network connectivity are the biggest limiting factors in mobile devices[3].

2.4.1. Application types

There are three application types for mobile devices: native, hybrid and mobile web application. Each of them has pros and cons which the developer should know about.

The applications which are programmed purely with specific languages for the specific mobile platforms are called native applications. For Android the language is Java and for iOS the language is Objective-C or Swift. Native applications have full access to all platform-specific application programming interfaces (API) and libraries. Programmer is able to use all the capabilities of the mobile device and usually native applications have better performance than other types of applications. Updates to native applications usually require some sort of approval process and rolling the update can take a long time[1][2].

Mobile web applications are actually websites optimized for mobile browsers. These applications or websites are accessed from the device's web browser. The access to the APIs and libraries in the device is very limited. Mobile web applications can be updated very easily, because it is enough to update only the server side. The offline capabilities of mobile web applications are very limited, because usually the network connection is required to access the server side[2].

Hybrib applications are a mix of native and web applications. The application is installed to the device and it launches as native application, but parts of the application's content comes from servers and are rendered in the device. It is also possible to write a hybrid application without writing native code at all, because there are frameworks[10] which can build the app for the developer from other languages[2].

[9]http://www.phonearena.com/news/What-defines-a-high-end-phonetablet_id34449
[10]http://mobile-frameworks-comparison-chart.com/

2.5. Mobile testing

Testing the mobile application thoroughly is very important, because the users can easily remove or change the application to another application.

79 percent of the users will remove the application if it does not work within the first two or three attempts. For 84 percent of the users the rating of the application in app store is important. So if your application has a bad rating in the app store, it will affect the downloads. The median expected load time for a mobile app to launch is 2 seconds[11].

2.5.1. Testing pyramids

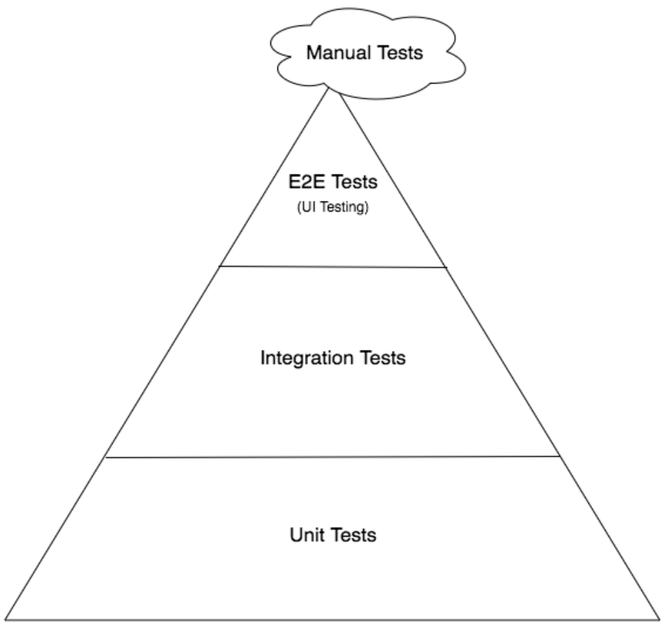


Figure 2: Traditional test automation pyramid, based on figure by Mike Cohn[2].

The traditional test automation pyramid (see Figure 2) was introduced by Mike Cohn[12]. Traditional test automation pyramid has three layers. From top to bottom they are end to end (E2E) tests, Integration Tests and Unit Tests. Manual testing is not part of the test automation pyramid so it is drawn as a cloud on top of the pyramid. The bottom of the pyramid is the widest and the top is the most narrow. The width of the pyramid presents the number of tests to be written in each layer[2].

Mobile test automation tools are not yet good enough to support the traditional test automation pyramid. Flipped testing pyramid (see Figure 3) is closer to the truth, because the tools cannot automate the tests as easily as with desktop applications. Mobile devices have a lot of sensors and other angles which the testing tools are not yet supporting or the support is not easy enough[2].

[11]http://offers2.compuware.com/rs/compuware/images/Mobile_App_Survey_Report.pdf
[12]https://www.mountaingoatsoftware.com/company/about-mike-cohn

Daniel Knott has created another pyramid which he calls the mobile test pyramid (see Figure 4). He has included manual testing and beta testing to the pyramid along with unit testing and E2E testing. Bolded layers, Unit Tests and E2E Tests, indicate the automated layers. Beta Testing and Manual Testing are manual layers. He claimed that by doing so it provides good results in projects[2].

This thesis focuses on the E2E testing, because that is the current layer where the improvement happens and where the need is the biggest.

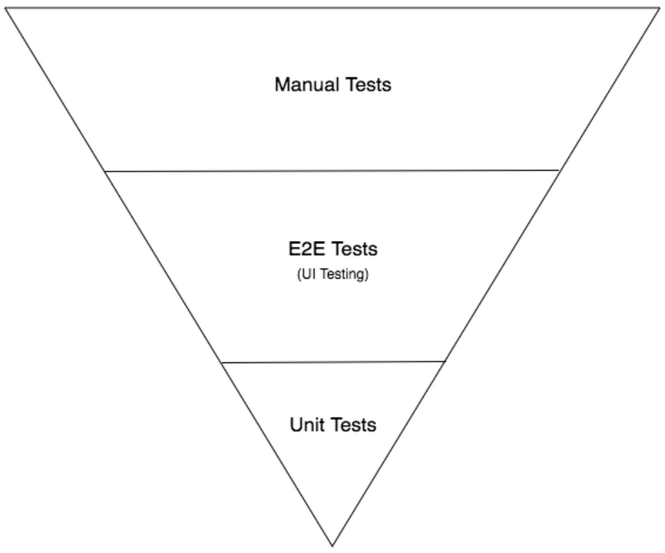


Figure 3: The flipped testing pyramid[2].

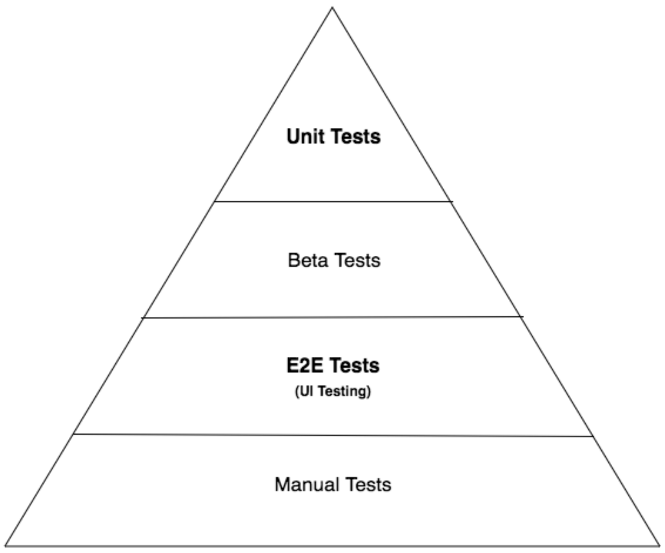


Figure 4: The mobile testing pyramid[2].

2.5.2. Test automation

Test automation means that a piece of software runs another piece of software and checks the output for a decision if the test passed or not.

If the automation is done properly it allows team to deliver high-quality code frequently[4].

Some reasons why one should use test automation are listed below[4]:

- Manual testing takes too long.
- Manual processes are error prone.
- Automation frees people to do their best work.
- Automated regression tests provide a safety net.
- Automated tests give feedback early and often.
- Tests provide documentation.
- Automation can be a good return on investment.

Automated tests can deliver results quickly, but automated tests can also take a long time to execute so it is hard to say if human tester is faster or slower than automated test. For example simple smoke tests or unit tests can be automated to be executed quickly. If the only function for human tester is to follow a predefined test plan without permission to diverge from the plan then the test automation beats human tester[4].

Humans are not good doing repetitive tasks, because they get tired and bored. Human nature makes the manual processes error prone[4].

Exploratory testing has been proven to be one of the best ways to find new bugs[5]. When repetitive tasks are automated the testers can focus their effort on more important work, such as exploratory testing[4].

Automated regression tests can provide a safety net. The programmer can get the feedback if the part of the system he touched broke something else in the system. Automated tests can be configured to give feedback early and often. For example it is possible to set tests to be executed on every save in the file and soon the programmer knows if the change broke anything the tests cover[4].

Tests with good coverage can be a good documentation of the system. To keep the tests passing the "documentation" has to be up to date. It is not mandatory to update static documentation to keep building the system and the documentation can fall back[4].

Test automation has also some downsides as well. Here is a list of problems or limitations test automation has[4][6][7]:

- Fragile
- Maintenance
- Automated tests rarely find new bugs
- Can be difficult

- Lose the focus
- Initial investment

Test automation can be fragile and the tests might break from seemengly trivial reasons. Test automation is sensitive to changes in behaviour, interface, data and context. Requirements of the system might change and any tests exercising the changed parts of the system will most likely be broken. Usually automated tests use some kind of interface to access the SUT and modification to that interface can break the tests. If the preconditions of the tests change it is called data sensitivity. Context sensitivity means the changes in things outside the system like network or hardware. Failure in these areas have the ability to break the tests[6].

Because test automation is fragile it needs maintenance and the test automation project should be treated like traditional software project[13][14]. Projects can have more test automation code than real application code and this leads to bigger maintenance burden.

Automated tests rarely find new bugs, because the variety of human actions in testing is the key to find new bugs in the software. The automation is good in finding reoccuring bugs in the software and this is called regression testing[7].

Test automation can be difficult. It depends on what one tries to automate. Unit tests are usually simple, but testing some bleeding edge system without good testing tools can be very difficult or nearly impossible. Software cannot be tested completely[5]. Sometimes it is hard to decide what should be automated and what should not.

Team might lose the focus when applying automation. If they add automation for automation's sake instead of focusing on improving the quality they are on wrong track[4].

Setting up the automated system can require bigger investment than just continuing the old manual testing. Usually the benefits will return the investment sooner or later. Test automation usually has a learning curve or "hump of pain" in the beginning (see Figure 5).

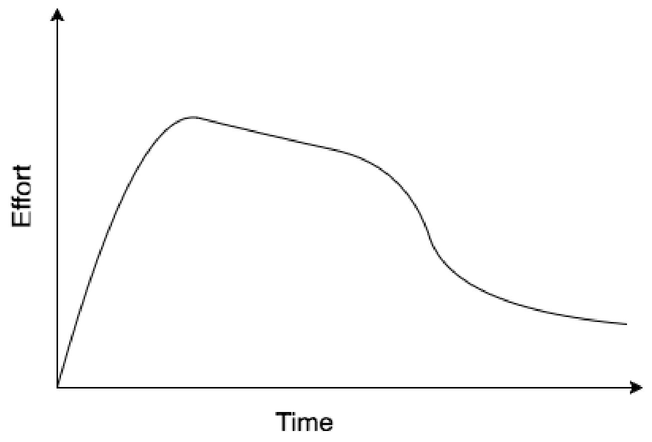


Figure 5: Learning curve or the "hump of pain".

[13]http://www.softwaretestinghelp.com/automation-testing-tutorial-7/
[14]http://www.methodsandtools.com/archive/archive.php?id=33

2.5.3. Different types of test automation tools

The main types of test automation tools are image, coordinate-based, OCR/text and native object recognition[2].

The tools which use image recognition try to find predefined graphical elements on the screen and act if it finds one. For example, if the developer has a set of icons he wants to click on the screen. He passes the icons to the tool which then tries to find the icons in the screen and click the coordinate where the icon was found. Image recognition is on it's best when the UI graphics doesn't change often. The image recognition itself is not platform dependant, because all it cares are the two images. These tests can be fragile if the predefined graphical elements are not carefully chosen. Badly chosen algorithms or algorithm parameters can also lead to flaky tests[2].

Coordinate-based recognition is not really recognizing anything on the screen. The test just blindly executes the given action on given coordinate. If the screen size varies between devices under testing the tests can be broken easily[2].

Optical character recognition (OCR) or text recognition is similar to image recognition, but the developer doesn't have to provide graphics to the tool. The developer inputs a text string to be found from the screen to the tool and it tries to find the string from the screen. The UI element must contain text to work with these kind of tools. OCR recognition tools tend to be slower than other types of tools, because they need to scan the whole screen for the text[2].

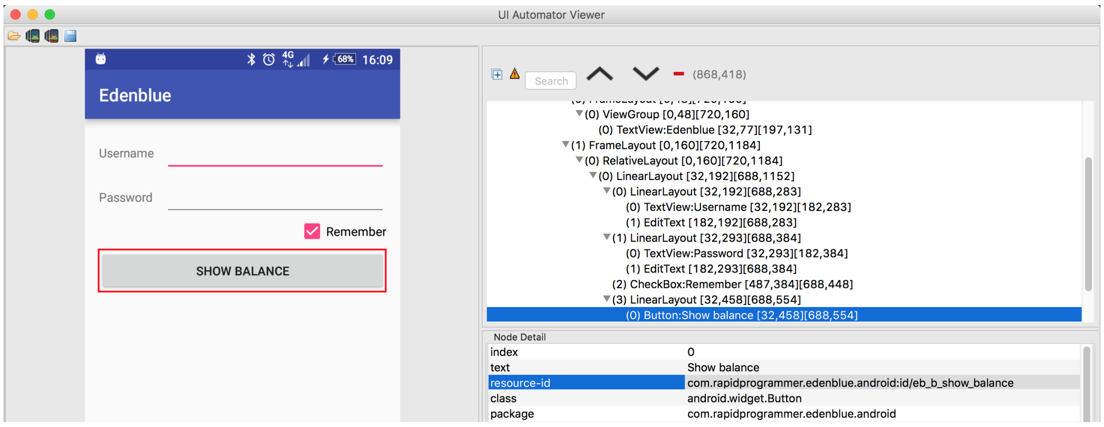
Native object recognition is the most widely used type of mobile test automation tools. These tools detect the UI objects with a UI element tree. There are many ways to access the UI elements, to name a few XML Path Language (XPath), Cascading Style Sheet (CSS) locators or the native object ID of the element. With native object recognition the developer can define the IDs or the locators properly and build very robust tests. The biggest advantage of this approach is that it doesn't depend on changes in the UI, orientation, resolution or the device itself[2]. In Figure 6 is shown Edenblue[15] application which can be automated easily with native object recognition. On the left there is the screen of the device, on the upper right corner the complete XML tree is shown and on the lower right corner there is the unique ID *eb_b_show_balance* for the *Show balance* button.

Many test automation tools are a combination of these types and they are not usually locked into single object recognition type. Every type has its pros and cons and the developer has to choose the best approach to his needs[2].

2.5.4. Record and replay

Record and replay (R&R), sometimes called as record and playback (R&PB) or capture and replay[8], refers to the fact that the developer can record his interactions with the application and later replay the same actions. R&R does not usually require programming skills and if the tool is good enough it is quicker to record the test than write the testing code for same scenario. Depending on the tool and the context R&R

[15]https://play.google.com/store/apps/details?id=com.rapidprogrammer.edenblue



```
▼(0) ViewGroup [0,48][720,160]
    (0) TextView:Edenblue [32,77][197,131]
▼(1) FrameLayout [0,160][720,1184]
  ▼(0) RelativeLayout [0,160][720,1184]
    ▼(0) LinearLayout [32,192][688,1152]
      ▼(0) LinearLayout [32,192][688,283]
          (0) TextView:Username [32,192][182,283]
          (1) EditText [182,192][688,283]
      ▼(1) LinearLayout [32,293][688,384]
          (0) TextView:Password [32,293][182,384]
          (1) EditText [182,293][688,384]
        (2) CheckBox:Remember [487,384][688,448]
      ▼(3) LinearLayout [32,458][688,554]
          (0) Button:Show balance [32,458][688,554]
```

index	0
text	Show balance
resource-id	com.rapidprogrammer.edenblue.android:id/eb_b_show_balance
class	android.widget.Button
package	com.rapidprogrammer.edenblue.android

Figure 6: Edenblue application can be automated with native object recognition[15].

tests can be even more fragile than regular tests and re-recording the same tests will decrease the gains of using R&R[6][9], see Figure 6.

In Figure 7 is described how R&R usually works. In production the UI is connected to the business logic directly. When the test is recorded the signals from the UI are intercepted by the Recording Decorator. When the signals are stored the decorator send the signals to the business logic and the AUT will continue as it would without the decorator. When the test is executed there is no need for the UI. Playback Driver reads the signals from the container and sends them to the business logic.

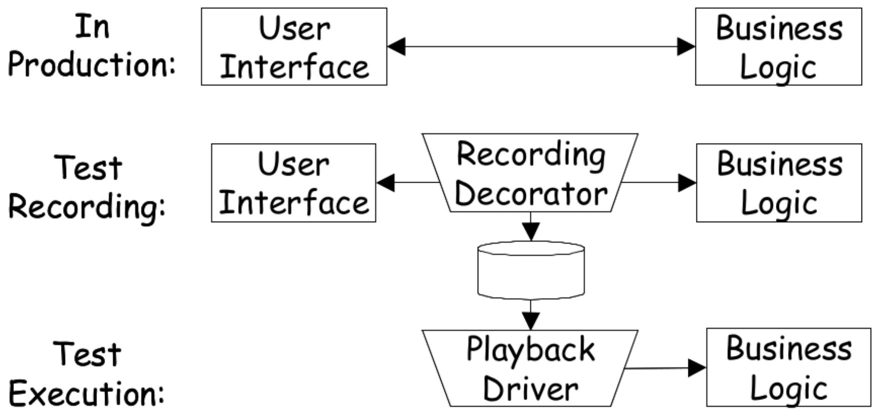


Figure 7: Record and replay using a recording decorator[6].

The developer should consider using R&PB tools in the following cases[6]:

- He needs to refactor a legacy system to make it amenable to XUnit-style hand-scripted tests and he feels it is too risky to do so without having regression tests.
- He cannot afford the time or cost of hand-scripting tests.
- He doesn't have the programming skills required to hand-script the tests.

The developer should avoid using R&PB tools in the following cases[6]:

- He cannot fix the behaviour of the system by freezing/snapshot the data on which the system will operate.
- The behaviour of the system is expected to change significantly between when the tests can be recorded and when they will be played back.
- If he wants to use the automated tests as a specification and there is no existing system that can be used for recording the tests.

2.6. Mobile game testing

Application quality is a real concern for companies around the world these days. If the application has low quality, it can trump good idea any day. This applies to mobile games as well. Crashes, glitches and slow response time, for example, will make gamers change the game to similar games from competitors, and it will lead to revenue loss. The main problem with games is that they utilize hardware usually a lot more than traditional apps and many games use direct screen access, in form of OpenGL or ActiveX, bypassing the OS level service. This section discusses more about these issues.

Traditional applications are usually meant to just make some task easier for the user and there are no other reasons why the user should use the application. Games don't have to be connected to the real life in any way. It is much easier to test the traditional application, because usually it works or it doesn't work as intended. It is much harder to measure if the game works or not, because the outcome is dependant on the user. The game can be the best game ever for one user and another user don't like it at all.

2.6.1. Difficulties in mobile games and test automation

Here is the list of the biggest difficulties of the mobile game test automation:

- Game launch times can be long
- Games take a lot of processing power
- Just rarely the objects in the game can be recognized natively
- Interpreting test result is even harder than in test automation usually[7]
- Automating the fun factor testing is impossible

Games have a lot of graphics and other assets which are required to run the game. Loading these assets can take a lot of time. If the median expected load time for a mobile app to launch is 2 seconds[16], it is very tough challenge for the game developers to load everything in time.

If the game has heavy graphics it might be that it is unusable with the low-end devices and could even be laggy in the midrange devices. Table 4 shows the low-end devices can be very slow with graphical games.

[16]http://offers2.compuware.com/rs/compuware/images/Mobile_App_Survey_Report.pdf

[17]The time is average time for five runs.

Table 4: Clash of Clans launch times for three devices with various hardware[17].

Device	Number of CPU cores	Memory (GB)	Launch time (s)
Samsung Galaxy S6	8	3	13
Sony Xperia Z3 Compact	4	2	19
LG Optimus L5 Dual E615	1	0.5	76

Automating mobile game testing is hard. If the game is not very simple, it is almost impossible to test the whole game with automated tools.

Games have hooks which are intented to make the player play the game again and again[10 p. 364]. Most of the hooks are inside the game and if the game does not even launch or run in the players device the hook will never be reached and the player will throw the game away even before playing the game.

When gamer downloads the game, he expects the game runs smoothly. The biggest mistake is to have a bug where the game crashes in the beginning. Smoke tests can be automated very easily.

Normal views in regular native applications are described usually in Extensible Markup Language (XML). The objects have unique identifiers which can be used to access the objects on the screen. Games runs usually on graphics containers like OpenGL container. The container is described in the XML, but the tools cannot access the objects inside the container. Currently the only way to access the objects in the game is to recognize the object location on the screen with image recognition or to program the container to expose the location of the elements inside of it.

In regular test automation the test result interpretation can be hard. In games it is even harder. Normal applications do not have the physics and randomization aspects the games have.

Fun factor is the gut feeling of the player he experiences when playing the game[10 p. 319-320]. Even if the game does not have any bugs and the idea is good, but the players do not feel the fun factor the game will not succeed. Because humans are different and another person might be entertained more by particular game than another person, it is impossible to automate the fun factor testing[18].

2.7. Regression testing

Regression testing is used to make sure new changes did not break the functionality which worked before. Regression testing is not another level of testing and it can occur at any level of test. Automated testing tools can support testing with this time-consuming task[11]. With games there might be a modification to scoring system and the regression test for it could be a scene where user exceeds 1000 points and he should win.

[18]http://www.gamedev.net/page/resources/_/creative/game-design/fun-factor-for-game-developers-r1828

2.8. Appium

Appium[19] is an open source test automation framework. Appium can test native, hybrid and mobile web applications on Android, iOS and Windows platforms.

One special feature of Appium is that the developer doesn't have to modify the application binaries to test the application, because Appium uses vendor-provided automation frameworks (see Table 5)[19].

Table 5: Appium vendor-provided automation frameworks[19].

OS	**Version**	**Automation framework**
iOS	9.3 and above	XCUITest
	9.3 and lower	UIAutomation
Android	4.2+	Ui Automator
	2.3+	Instrumentation
Windows	All	Microsoft's WinAppDriver

Appium uses WebDriver[20] protocol to wrap the vendor-provided framework into a single API. WebDriver specifies a client-server protocol (known as the JSON Wire Protocol[21]) for the communication. The clients have been written in many major programming languages like Ruby, Python and Java[22].

Appium sets up server into the host machine. The client, where the test logic is located, connects to the server. If the operating system of the device is Android, the server forwards the commands from the client to the device via Ui Automator framework (Figure 8). On older Android devices the server communicates with the device via Selendroid (Android API level < 17). If the operating system in the device is iOS, the server forwards the commands via XCUITest (iOS 9.3 and above) or UIAutomation framework (iOS 9.3 and lower) (Figure 9)[23].

2.9. Image recognition

The process of identifying and detecting an object or a feature in a digital video or image is called image recognition. This concept is used in many applications like systems for factory automation, toll booth monitoring, security surveillance and lately even in mobile applications.

[19]http://appium.io/introduction.html
[20]http://docs.seleniumhq.org/projects/webdriver/
[21]https://code.google.com/p/selenium/wiki/JsonWireProtocol
[22]http://appium.io/downloads
[23]http://www.slideshare.net/saucelabs/appium-basic-20296603

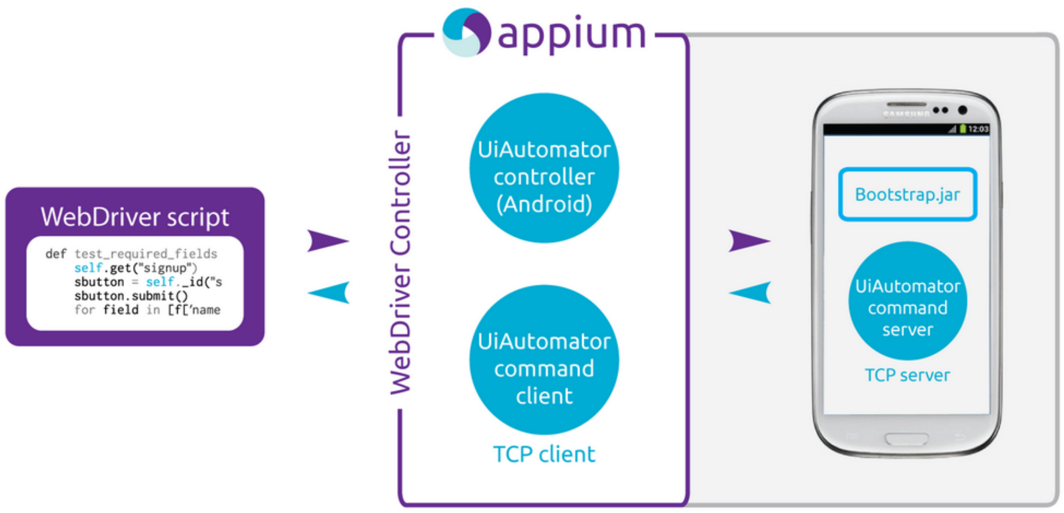


Figure 8: Appium on Android architecture[23].

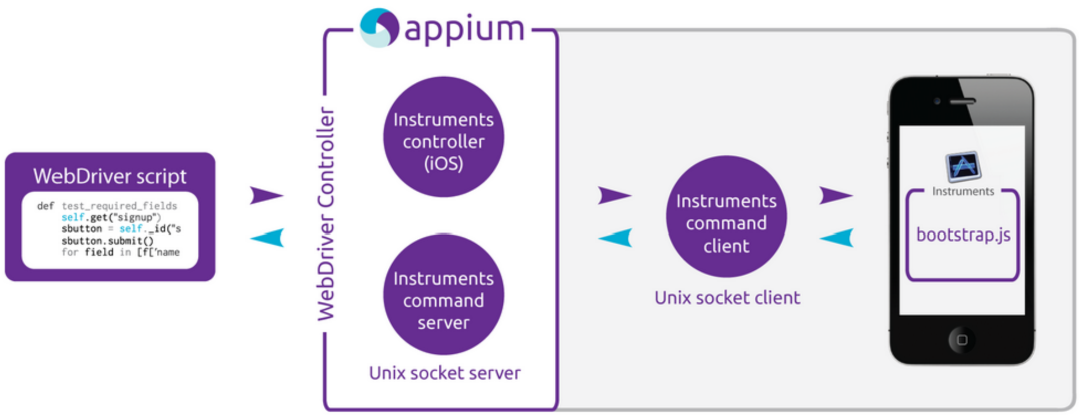


Figure 9: Appium on iOS architecture[23].

2.9.1. OpenCV

OpenCV[24] is an open source computer vision library which includes several hundreds of computer vision algorithms. It was originally developed by Intel[12]. It has C, C++, Java, Python and MATLAB interfaces and it supports Linux, Android, Mac OS and Windows.

2.9.2. Image features

A feature is *something that can be measured in an image*. Object can be characterized by a single feature, but it is more usual to use a set of features. In object recognition the first problem is to find a part of the image that might contain an object[12]. To detect

[24]http://opencv.org/

and locate an image in another image one needs to find the features in the image. In Figure 10 the boxes contain flat surface (1), edge (2) and corner (3). Flat surface can be a feature, but it is not a good one because one cannot locate the feature in another image. Edge can also be a feature and it is better than flat surface, but still the location is unclear. In Figure 10 the edge is locked to y-axel, but one cannot tell what is the x-coordinate. The corners are the best features, because the coordinate of the feature can be found easily[25].

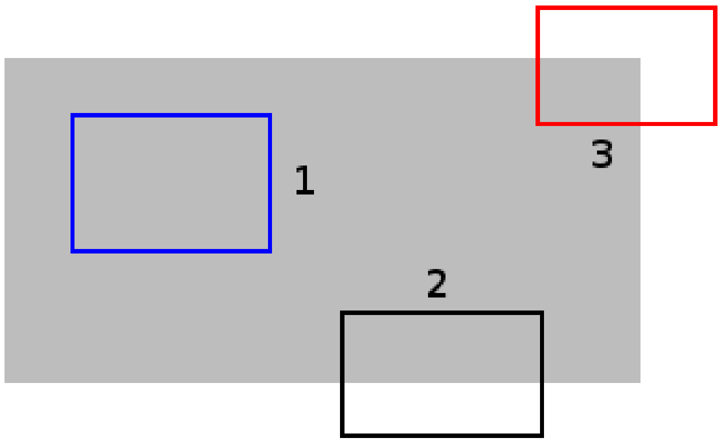


Figure 10: Flat (1), edge (2) and corner (3) features.

2.9.3. Rotation invariant

If the method is rotation invariant it can still find the same features when the image is rotated (Figure 11)[25].

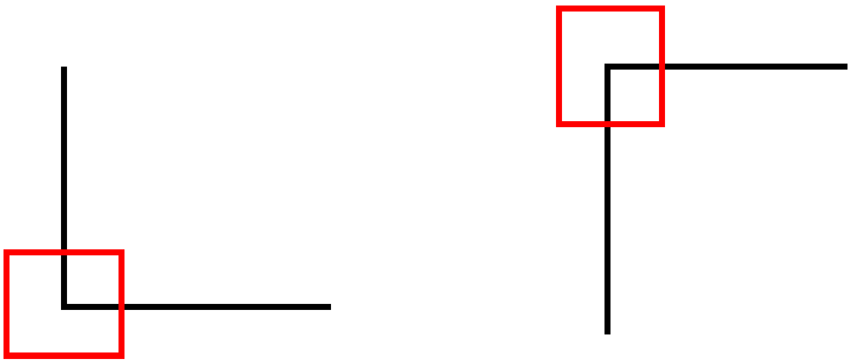


Figure 11: Left is original and right is rotated.

[25]http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html

2.9.4. Scale invariant

If the method is scale invariant it can still find the same features when the image is scaled. In Figure 12 the image is scaled in and the corner begins to look more like curve. Scale invariant method can return the same feature from both images[25].

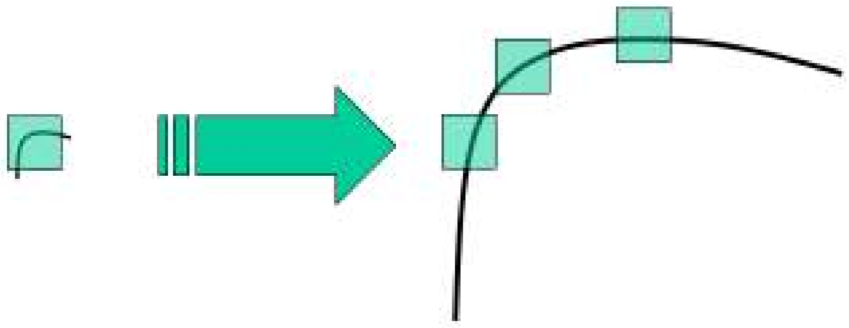


Figure 12: Left is original and right is scaled[25].

2.9.5. SIFT and SURF

Scale Invariant Feature Transform (SIFT[13]) and Speeded Up Robust Features (SURF[14]) are widely used algorithms to detect image in another image. SIFT and SURF blurs the image to reduce noise, but at the same time the edges and corners drift away from the original location[15].

2.9.6. KAZE

KAZE features are a novel multiscale 2D feature detection and description algorithm in nonlinear scale spaces. KAZE does not use Gaussian blur to reduce the noise, but it makes blurring locally adaptive to the image. This reduces the noise and preserve the original edge and corner locations. KAZE features are somewhat more expensive to compute than SURF, but compared to SIFT KAZE features have better performance and detection[16]. KAZE uses Additive Operator Splitting (AOS) schemes to solve the nonlinear diffusion equation[17].

2.9.7. AKAZE

Accelerated KAZE features (AKAZE) are improved version of KAZE features. Calculating AOS in KAZE is computational intensive. Instead of AOS Fast Explicit Diffusion schemes (FED), are used in AKAZE to speed up the feature detection in nonlinear scale spaces. AKAZE also introduces Modified-Local Difference Binary (M-LDB) to preserve low computational demand and storage requirement. AKAZE is faster to compute and gives better results than previous methods like SURF, SIFT and KAZE[17].

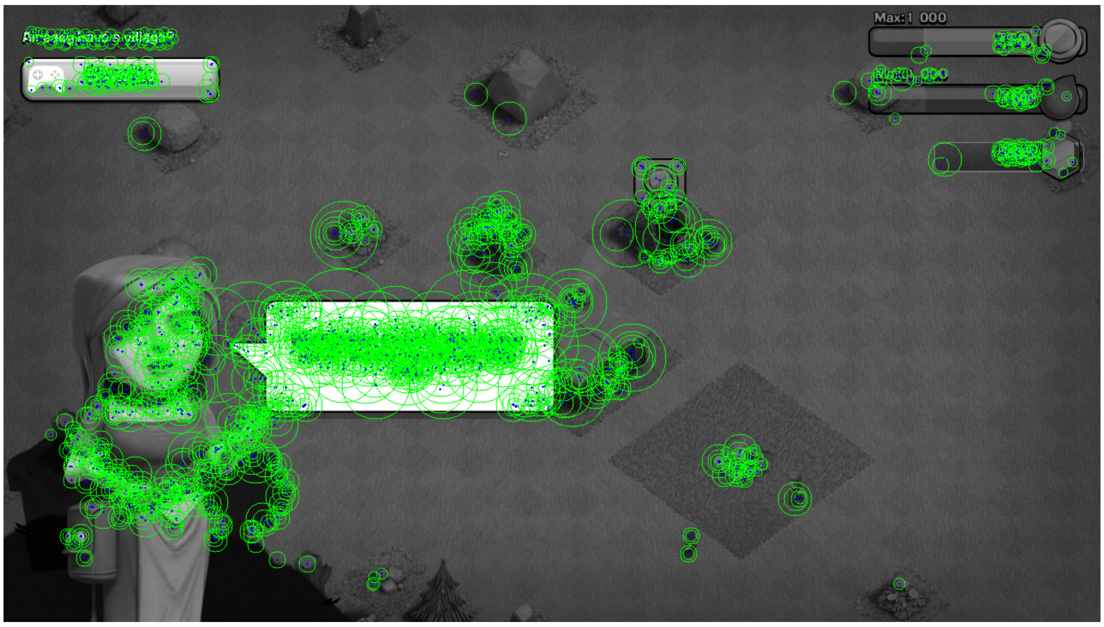


Figure 13: AKAZE features in Clash of Clans screenshot

2.10. Tools

Virtual Network Computing (VNC): is a system which is able to control another computer remotely over a network connection. Keyboard and mouse events as well as graphical screen updates are transmitted from one computer to another.

Android Debug Bridge (ADB)[26]: is a toolkit for Android development. ADB can control emulator intances or real device from command line. It is a client-server program which consists of three components, client, server and daemon. Client runs on the development machine and it is connected to the server. Server runs also in development machine and is connected to the daemon which runs in the target instance.

pyadb[27]: is a Python wrapper for ADB. User can call ADB commands from Python code.

Bottle[28]: is a Web Server Gateway Interface (WSGI) micro web-framework written in Python. It is designed to be simple, lightweight and fast. Bottle does not have other dependencies other than the Python Standard Library and it is distributed as a single file module.

[26]http://developer.android.com/tools/help/adb.html
[27]https://github.com/sch3m4/pyadb
[28]http://bottlepy.org/docs/dev/index.html

2.11. Related application testing tools

This section compares existing application testing tools that might be able to solve the problems stated in section 2.6.1 and chapter 3.

SikuliX[29]: automates everything on the screen. The former name for SikuliX was Sikuli[30]. It uses OpenCV image recognition to find the objects to click on the screen. SikuliX doesn't have support for mobile devices out of the box, but it is possible to make it work with simulators, emulators or VNC solutions where the mobile device screen can be accessed from the desktop[18][19].

Testdroid Recorder[31]: is a free plugin for Eclipse[32]. It is a record and replay tool (see Section 2.5.4). Testdroid Recorder records user actions with the application under testing (AUT) and generates reusable Android JUnit, Robotium[33] and ExtSolo[34] tests. The generated tests can be replayed afterwards.

Robotium Recorder[35]: is a commercial plugin for Android Studio, very similar to Testdroid Recorder and it can also record and replay Robotium tests.

Appium GUI[36]: is a project which provides graphical user interface (GUI) for Appium (see Section 2.8). There is an inspector which can tell information about the objects on the screen and also a recorder which can record and replay Appium tests.

JAutomate[37]: is a commercial tool combining image recognition with record and replay functionality[9]. JAutomate does not support mobile devices out of the box, but it is possible to make it work with simulators, emulators or VNC solutions where the mobile device screen can be accessed from the desktop.

Table 6: Related application testing tool comparison.

	Desktop support	Mobile support	Image recognition capabilities
SikuliX	X		X
Testdroid Recorder		X	
Robotium Recorder		X	
Appium GUI		X	
Jautomate	X		X
MAuto		X	X

[29]http://www.sikulix.com/
[30]http://www.sikuli.org/
[31]http://testdroid.com/products/testdroid-recorder
[32]https://eclipse.org/
[33]https://github.com/RobotiumTech/robotium
[34]https://github.com/bitbar/robotium-extensions
[35]http://robotium.com/products/robotium-recorder
[36]https://github.com/appium/appium-dot-app
[37]http://jautomate.com/

2.12. Summary

This chapter has explained the essential terms, techniques and concepts for the reader to understand the concepts behind the tool which will be proposed later in this thesis. We have shown how the mobile industry has grown over the past years and how the growth has caused fragmentation effect which makes it harder to provide a quality software for the end user. Mobile application testing is younger topic than desktop application testing and this chapter has shown the main differences between mobile and desktop application testing and test automation. By reading this chapter the user has understood what is the difference between application testing and test automation as well as the difference between mobile application testing and mobile game testing. Image recognition is a key component in the tool this thesis proposes so the basics and algorithms are explained in this chapter.

3. PROBLEM STATEMENT

In manual testing the human user gives the input to the application under testing and verifies if the output is expected or not. To automate this the human user needs to be removed from the equation and there needs to be a way to give the input to the application and verify the output from the application programmatically. Current tools for mobile application testing are focusing on native object recognition to check the integrity of the developed functionality. However, mobile games require other communication means to transmit the events back and forth, because the native object recognition does not work for most of the mobile games (see Figure 14).

Usually, the functionality of a mobile game is executed during runtime in a graphic container, e.g., OpenGL, to provide better graphics and physics for users. The container wraps all the functionality of a game. Thus, it is not possible to access that wrapped functionality to test it. Several methods have been developed to overcome this problem. Most common and effective techniques are 1) programming the container in a particular way to expose functionality outside the container and 2) implementing image recognition approaches to identify functionality from the screen to transmit this functionality to the testing process. Generally, image recognition is more simpler to use. By using image recognition methods, it is possible to find and locate specific image locations, e.g., subimage.

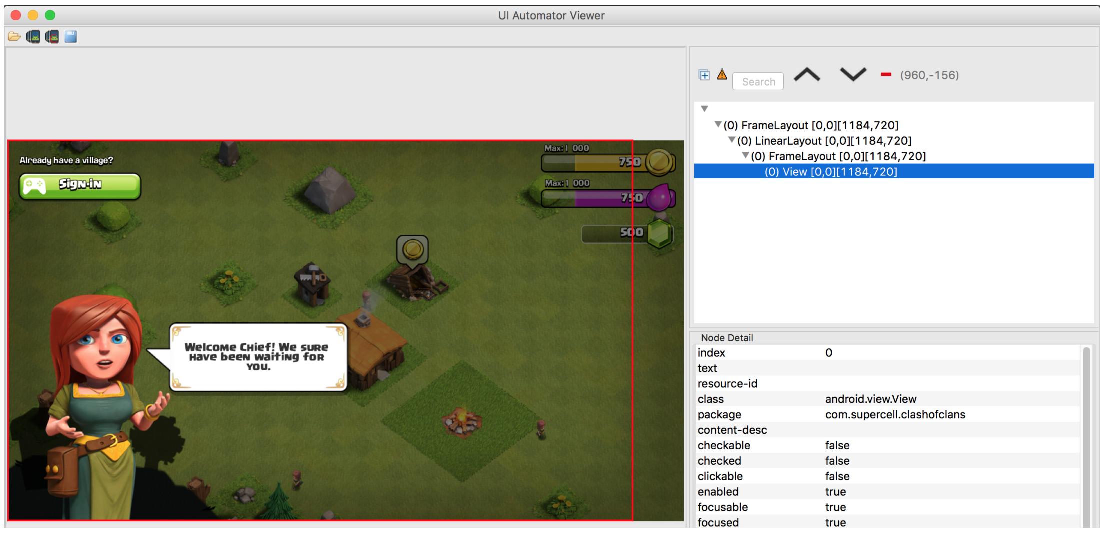
To use image recognition the user needs the graphical representation of the object to find, e.g., buttons or game characters. Sometimes the user can get the elements directly from the graphics designer, but this is not always the case. Also the game might change the environment and context where the object is presented, e.g., shadows and lighting. This will affect the success ratio of the image recognition, so it is better to use the actual context from the game and take screenshots while playing the game.

To extract an object from the game in real context the user needs to take a screenshot while the game is running in the mobile device. The screenshot is stored in to the memory of the mobile device so the user needs to transfer the image to his machine. Once the screenshot is available in the user's machine, the object or partial image must be extracted from the screenshot. Once every object required to run the game programmatically are extracted the user needs to use these objects and write the automation code to replay the sequence he played before.

To make the cycle above easier and faster for the user we propose a tool called MAuto. MAuto will automatically take the screenshots and extract the objects from those screenshots while the user is playing the game. Once the sequence is ready, MAuto will generate Appium test code to replay the sequence. We will discuss about MAuto in more details in the next chapter.

3.0.1. Methodology

As already pointed, MAuto employs an image-recognition based methodology in order to identify whether the icons identified in the graphical display of the mobile screen are coherent with game functionalities. For this purpose, the following are part of key attributes of MAuto:



```
▼
 ▼(0) FrameLayout [0,0][1184,720]
   ▼(0) LinearLayout [0,0][1184,720]
     ▼(0) FrameLayout [0,0][1184,720]
         (0) View [0,0][1184,720]
```

Node Detail

index	0
text	
resource-id	
class	android.view.View
package	com.supercell.clashofclans
content-desc	
checkable	false
checked	false
clickable	false
enabled	true
focusable	true
focused	true

Figure 14: The lowest native object in Clash of Clans is View.

Query image is the image cropped from a screenshot and which is used to find and element from another screenshot in the replay phase.

Native object recognition finds the object from UI element tree.

Image recognition finds the object from video frame in the device screen.

Traditional mobile application is an application which is not a game and they don't use OpenGL, ActiveX or similar tools to "hide" the objects from native object recognition.

Mobile application is a game played on mobile device, f.e. Clash of Clans.

Recording phase is the phase where the tester uses the application under testing to record the inputs for later replay phase.

Replay phase is the phase where the earlier recorded test is rerunned on the same or other device.

Stable test is a test which doesn't give false results and it does what is intented. Vice versa **Unstable test** is a test which can fail even if the application worked as expected or pass even if the application had some unexpected behaviour. Tester can rely more on stable test and unstable tests can make the test result investigations harder, because the problem could be in the system testing the application, not in the application itself.

4. QUESTIONNAIRE

Five test automation experts answered to this questionnaire. The purpose of this questionnaire was to find the common problems with the test automation on mobile games and see if the literature correlates with the real world.

The summary of the questionnaire results is that most of the experts use Appium as their testing tool/framework. The thesis already highlighted why it is harder to automate mobile games than mobile applications and the experts agree with this. They think it is hard to create good query images to make the tests stable. There were not common problems with replaying the tests on mobile devices, but they agreed that there are wide variety of issues to overcome, also in the replay phase. They would use test automation for repetitive tests and human testers for more intuitive testing. In their opinion, the most time consuming issue in test automation is to make the tests stable on wide variety of devices.

The questions raised to the experts are the following:

- What is your preferred framework/tool to create automated tests on Android?
- How easy it is to automate mobile applications which are not games?[1]
- How easy it is to automate mobile games?[1]
- What are the biggest issues with game test automation and why?
- What possible problems should be considered when doing image recognition based test automation?
- What kind of image is a good query image for games (query image = the image to be found from the screen in replay phase)? Does lighting or shadows matter?
- What are the biggest issues in the replay phase of testing when testing non-game applications?
- What are the biggest issues in the replay phase of testing when testing games?
- When the tester should prefer automated tests instead of manual tests?
- When the tester should prefer manual tests instead of automated tests?
- What are the most time consuming tasks when creating automated tests (this does not include test replay on other devices)?

4/5 of the experts preferred Appium (Section 2.8) as their primarily testing framework/tool. One of the experts preferred Robot Framework over Appium. We chose Appium as well with MAuto, because we think it is the best framework to create automated tests which work on many different devices and it can be easily modified and extended.

The experts think it is harder to automate mobile games than other applications, because majority say it is hard to automate mobile game and just one says the same when it comes to applications which are not games.

[1]A value between 1 and 5.

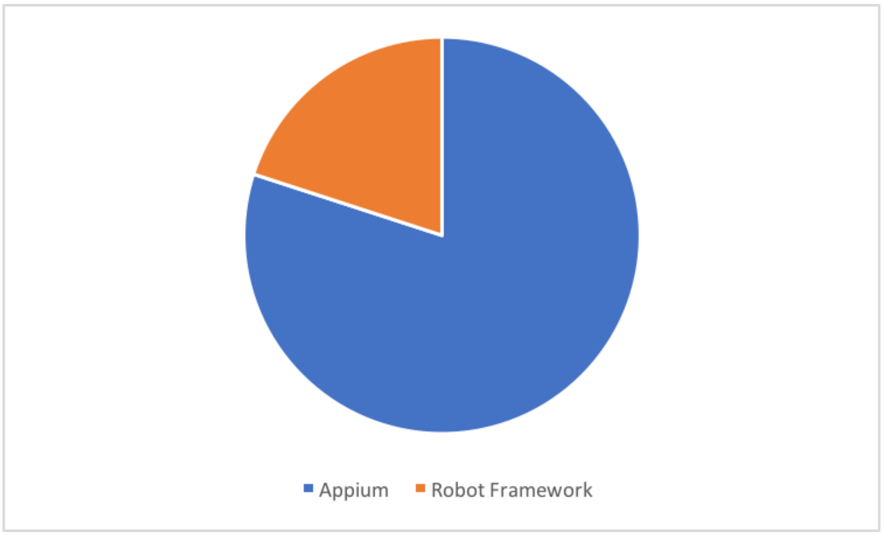


Figure 15: Preferred frameworks.

Standard deviation for Figure 16 is 0.707106781 (scale 1-5) and 0.447213595 (scale 1-3).

Standard deviation for Figure 17 is 1.095445115 (scale 1-5) and 0.707106781 (scale 1-3).

4.1. The biggest issues in game test automation

From the questionnaire, we noticed that 3/5 of the experts think it is hard to get the game to specific state.

Similarly, 3/5 of the experts think the games are hard to automate, because most of the games won't expose the elements in the game. This is one of the main reasons we decided to create MAuto.

4.2. The problems with image recognition based test automation

The problems are mostly related to good query images. It is hard to find a query image which works in different devices. It is also hard and time consuming to maintain a good query image catalog. When there is a change in the UI of the application, it might break many automated test sets. In the worst-case scenario, those tests must be recreated from the scratch and it is very time consuming.

The experts agree that the best query image is any image that makes the test work and it should preferably work on multiple different devices. The problem just is that the query image that works depends on the situation. Sometimes very small or very strictly cropped query image is the best and sometimes the image doesn't have to be like that. Sometimes it is a good idea to use some surrounding object and pass the input event coordinates relatively to this surrounding object.

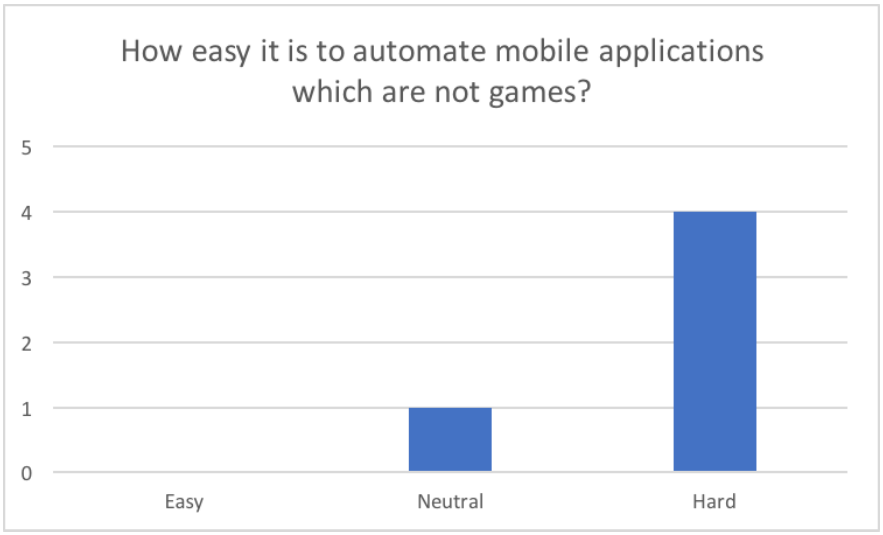


Figure 16: The results for question "How easy it is to automate mobile applications which are not games?"

4.3. The biggest issues in the replay phase of testing

The experts didn't have much common problems, but most of the problems were related to screen sizes and resolutions or random visible events like “battery is almost empty”-popups. Some of the experts thought the timing of the inputs are sometimes problematic. For example, if an element is only visible a short amount of time, it is hard to detect the element and interact with it.

4.4. Test automation vs. manual testing

The experts agree that the automated tests should be used to test repetitive tasks and things that are tested on each application version (regression tests). Manual testing is preferred when the testing should be more intuitive.

4.5. The most time consuming issue in test automation

Most experts thought it takes a lot of time to stabilize the tests. For example, if a new test step is added to the end of a long test set, the tester needs to run the whole set to see if the added step works or not. Once it is running on a single device there might be a lot of errors on other devices.

4.6. Conclusion

This questionnaire shows that it is not easy to create automated tests for a wide variety of devices. Many of the problems comes from the device fragmentation. Random

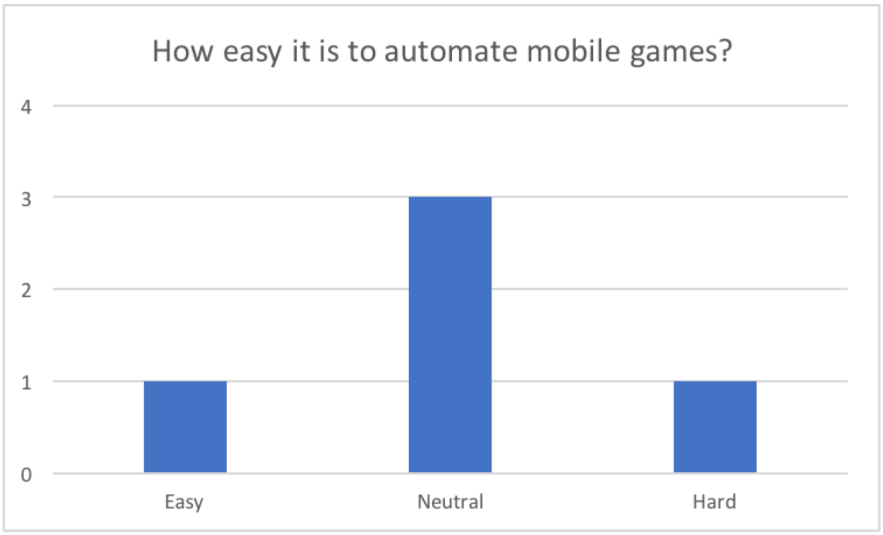


Figure 17: The results for question "How easy it is to automate mobile games?".

events also cause a big headache to test experts when the number of tests and devices increase.

5. MAUTO: EASIER MOBILE GAME TEST AUTOMATION

The purpose of MAuto is to make it easier to create image recognition based mobile test automation and this section describes how MAuto is designed and implemented. Section 5.1 shows the high level system design and section 5.1.1 digs deeper into the system. In MAuto AKAZE is used to recognize the objects from the screenshots and this is covered in section 5.2. MAuto is used to generate automation code for Clash of Clans in section 6.1.

5.1. System overview

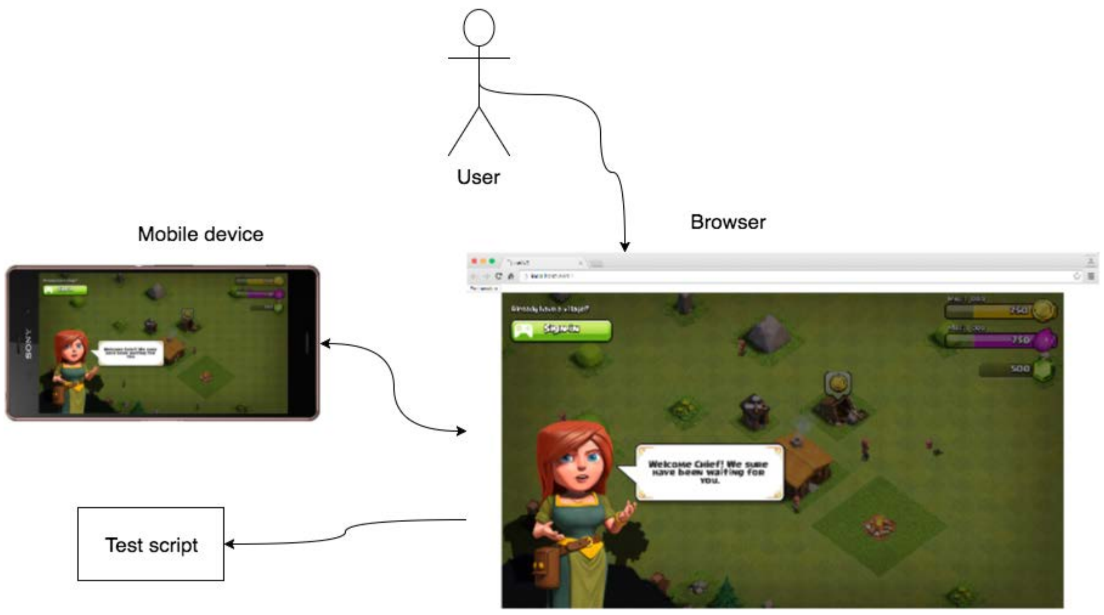


Figure 18: System overview.

We propose a tool called MAuto which helps the user to create Appium tests with image recognition without coding a single line of code. The main target is to help developer to test mobile games, but the tool can be used for other application types as well.

MAuto is designed to be an easy to use mobile test creator. The system consists of three elements, the user, the browser and the mobile device and it generates test sript which the user can run later (see Figure 18). Once the user has launched MAuto all the interaction between the user and the tool happens with the browser. MAuto takes care of the mobile device and all the user needs to do is to plug in the USB cable, start MAuto and interact with the application via web browser. When the user is done with the recording MAuto will generate a test script which is able to reproduce the recorded events. MAuto itself is not able to replay the test, but the test script can be replayed with Appium.

5.1.1. Architectual design

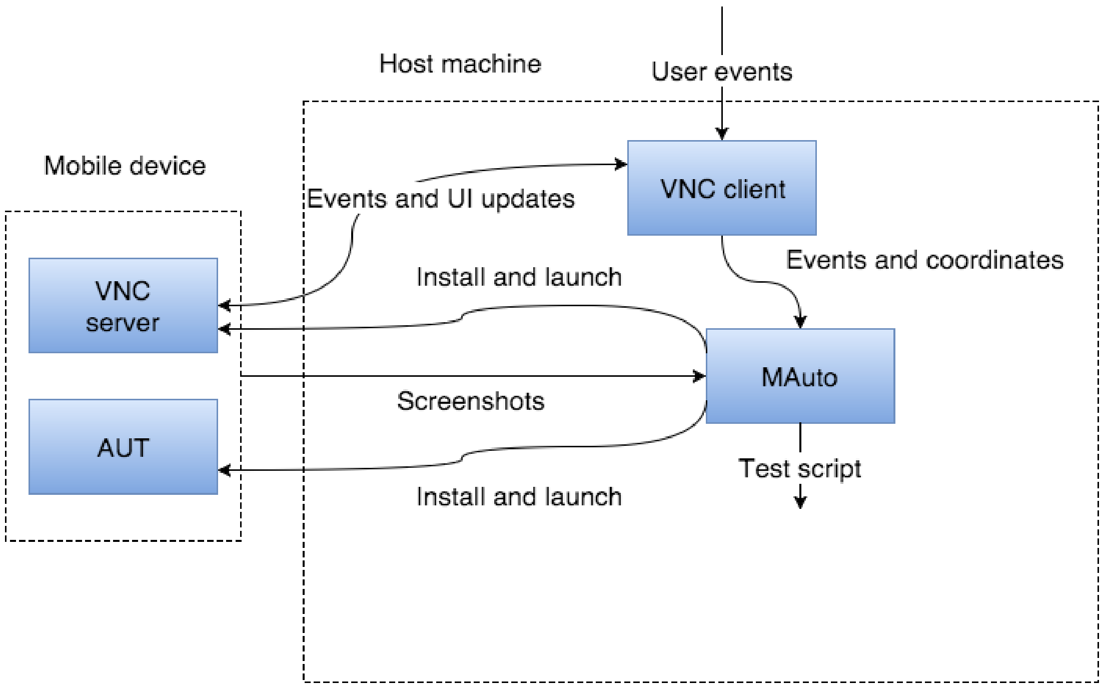


Figure 19: Architectual design.

In Figure 19 is described a more detailed view of the system. The two physical components are mobile device and host machine. In the beginning MAuto installs and launches the application under test (AUT) and the VNC server to the device. Then MAuto prepares the connection between VNC server and client. On event from the user VNC client forwards the event with the coordinates to MAuto. MAuto takes the screenshot from the mobile device, saves the screenshot and gives VNC client permission to continue. VNC client sends the same event to VNC server and the view from the mobile device will be updated to the VNC client. When the user is finished MAuto will generate the test script from the screenshots and events.

In Figure 20 is shown how the recording sequence goes with MAuto. At first the user launches the MAuto from command line and gives the path to the AUT. First MAuto will install the VNC client to the device and launch it. Then it will install the AUT and launch it. MAuto will run a webserver which the browser can access and when everything is ready for the recording MAuto will open the webpage to the browser. The user can now see the screen from the device in the browser. Then the user begins to give the events to the VNC client running in the browser. The event goes first to the VNC client. MAuto runs modified VNC client which will send the event with coordinates to MAuto. MAuto will save the event, take a screenshot from the screen of the mobile device and extract the query image around the event coordinates.

The API call from the VNC client to MAuto will return after the images have been processed. Then the VNC client will pass the event to the device via VNC protocol and the UI in the VNC client will be updated. This will continue until the user decides to stop the recording. Finally when MAuto gets the command to stop the recording it generates the test script which can be used with Appium to replay the test on any given device.

5.1.2. Record and replay

My tool is a R&R tool. The tool records the user interactions and Appium can be used to replay the tests. The recording decorator is modified VNC viewer in the browser and replay driver is Appium test with image recognition (see Figure 7).

5.1.3. Generated code

MAuto stores the screenshots and query images to session folder. That folder also has a CSV file where the events and image name are saved. See example CSV file in Listing 5.1.

Listing 5.1: Example clicks in CSV format.

```
354;553;screenshot-1.png;cropped-1.png
535;314;screenshot-2.png;cropped-2.png
180;391;screenshot-3.png;cropped-3.png
176;388;screenshot-4.png;cropped-4.png
```

Test script generator loads the CSV file from the disk and transforms it to Appium compatible test file. The generated test consists of the following sections.

1. Python imports
2. Helper functions
3. Driver initialization
4. Test clicks
5. Driver quit

Test click looks like this

```
wait_click("/Users/jarno/projects/dcode/sessions/0556ffef/screenshots/cropped-6.png")
```

5.2. Image recognition with AKAZE

AKAZE was chosen to detect the query image location in the device screen, because it is better than widely used SIFT and SURF[1][17] and it is free to use[2].

We calculate AKAZE features in the query image and the current screenshot. Once both features are calculated, we will compare those features to detect if the query

[1]http://www.robesafe.com/personal/pablo.alcantarilla/kaze.html

[2]https://github.com/pablofdezalc/akaze/blob/master/LICENSE

image is currently shown on the screen and what are the coordinates. This is shown in the Figure 21. The green circles are the calculated features and the red lines are the matching features in both images. Once we have the matching features, we calculate the average coordinate from the inliers to get the coordinate of the query image in the screenshot.

5.3. Summary

This chapter has proposed the tool called MAuto to make it easier and faster for the user to create mobile test automation scripts using image recognition. MAuto is a tool which records user inputs and replays those inputs on any Appium supported Android device. This is the main sequence: User clicks a coordinate in the browser, MAuto detects the click, takes a screenshot, crops query image around the coordinate and generates Appium code that can be replayed. In the replay phase Accelerated KAZE features are used to find the query images from the device screen.

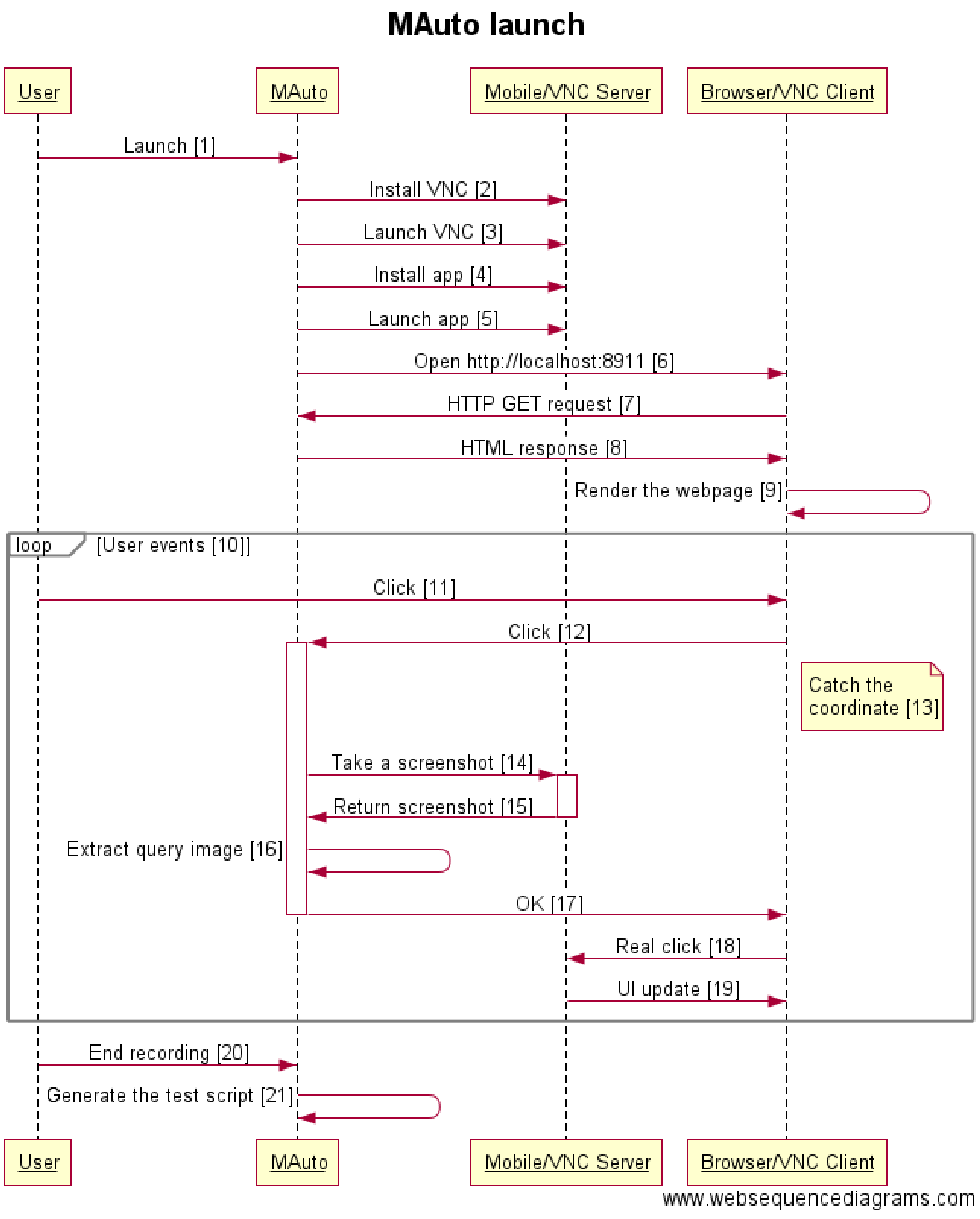


Figure 20: Sequence diagram.

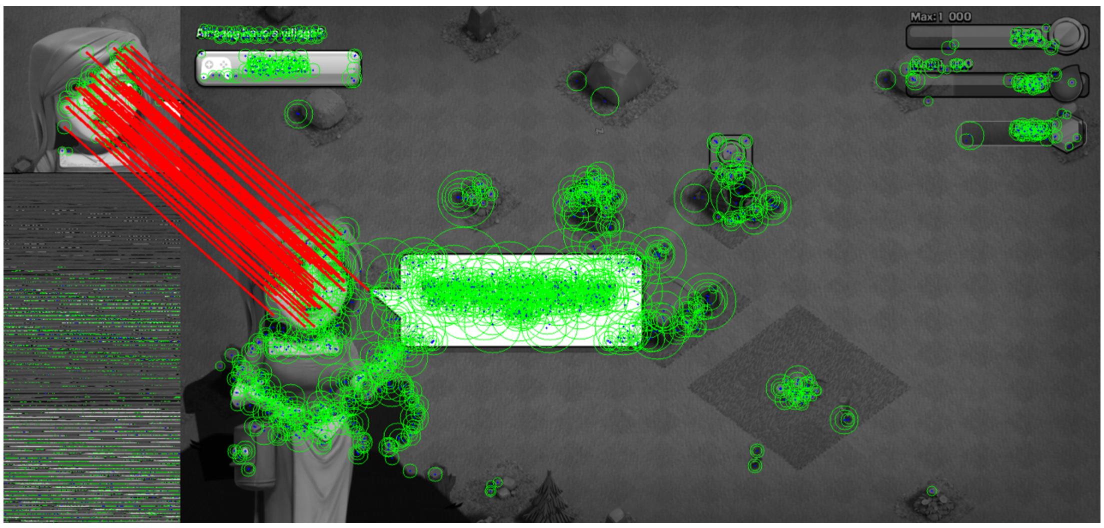


Figure 21: Matching AKAZE features in the query image and screenshot.

6. VALIDATION AND RESULTS

6.1. Use Case — Clash of Clans

This section shows how MAuto is used in real environment with real mobile game. The use case shows how to automate the tutorial for Clash of Clans Android mobile game[1] (version 8.551.4) from Supercell[2].

Clash of Clans (CoC) is a mobile Massive Multiplayer Online Game (MMO/MMOG) where the player builds a community, trains troops and attacks other players to earn assets.

The game has a tutorial which the player must pass to play the game. If this tutorial can be passed without serious bugs, it is a good indicator that the game works propely without any fatal bugs.

The tutorial tells the player to click certain elements to continue. The variation is very limited in the tutorial so it should be perfect test subject for MAuto.

Launch MAuto recording

```
python start-server.py --apk clash-of-clans-8-551-4.apk --package com.supercell.clashofclans --activity GameApp
```

The browser (see Figure 22) pops up and we are ready to begin recording.

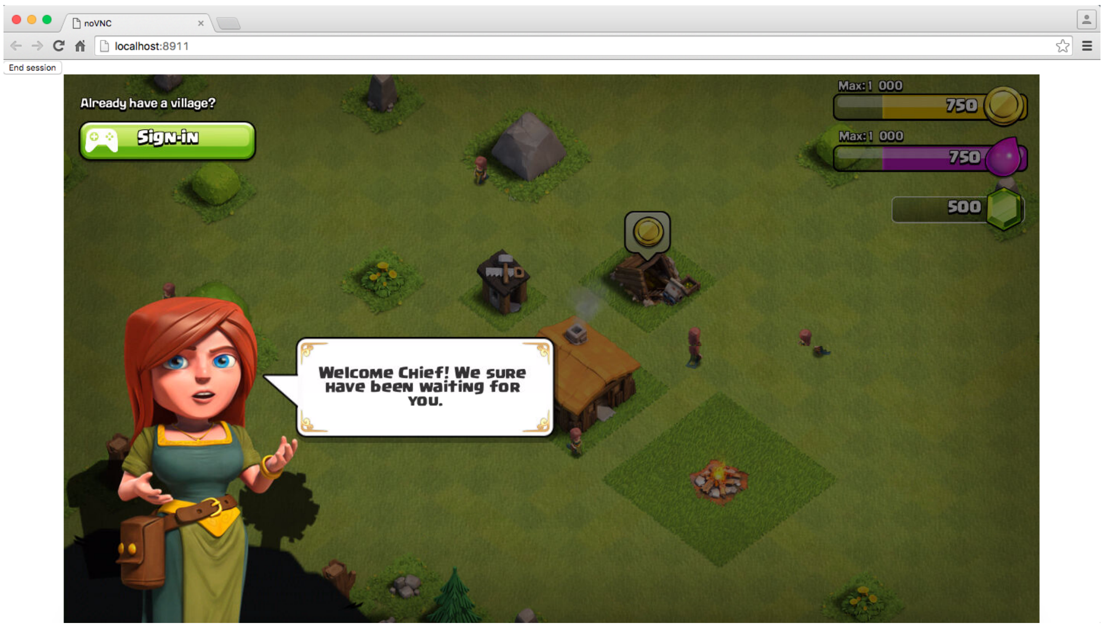
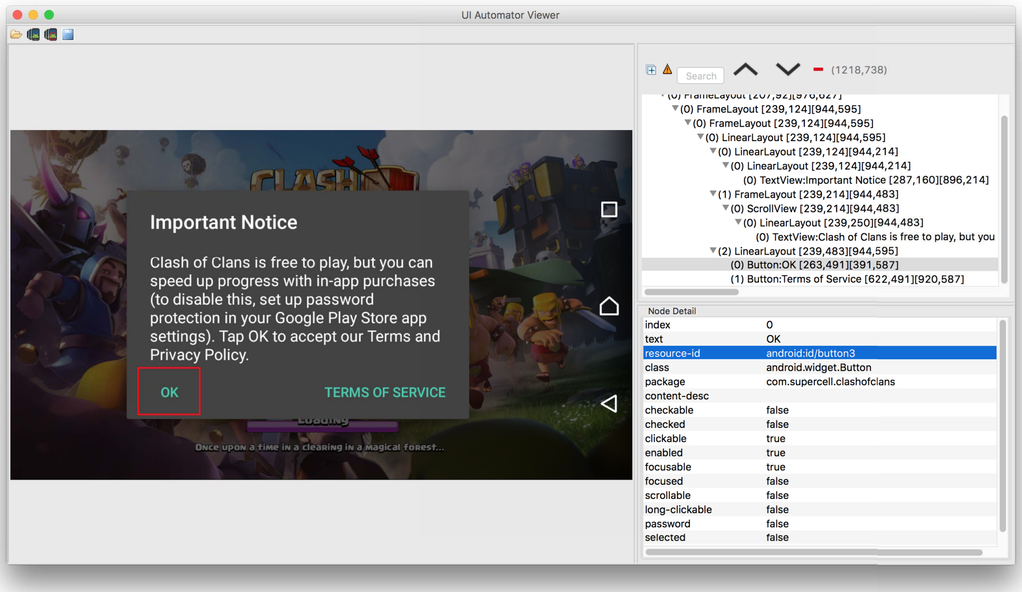


Figure 22: Browser looks like this with Clash of Clans.

The first view which requires user interaction in Clash of Clans is the *important notice* view. This view can be accessed with native object recognition which usually better than the image recognition (see Figure 23). We can see the object's resource id is *android:id/button3* and the package is *com.supercell.clashofclans*.

Clash of Clans requires the user to select one Google Play account for the game. This view can be accessed with native object recognition as well (see Figure 24).

[1]https://play.google.com/store/apps/details?id=com.supercell.clashofclans
[2]http://supercell.com/



Important Notice

Clash of Clans is free to play, but you can speed up progress with in-app purchases (to disable this, set up password protection in your Google Play Store app settings). Tap OK to accept our Terms and Privacy Policy.

OK TERMS OF SERVICE

```
(0) FrameLayout [207,92][976,627]
  (0) FrameLayout [239,124][944,595]
    (0) FrameLayout [239,124][944,595]
      (0) LinearLayout [239,124][944,595]
        (0) LinearLayout [239,124][944,214]
          (0) LinearLayout [239,124][944,214]
            (0) TextView:Important Notice [287,160][896,214]
        (1) FrameLayout [239,214][944,483]
          (0) ScrollView [239,214][944,483]
            (0) LinearLayout [239,250][944,483]
              (0) TextView:Clash of Clans is free to play, but you
        (2) LinearLayout [239,483][944,595]
          (0) Button:OK [263,491][391,587]
          (1) Button:Terms of Service [622,491][920,587]
```

Node Detail

index	0
text	OK
resource-id	android:id/button3
class	android.widget.Button
package	com.supercell.clashofclans
content-desc	
checkable	false
checked	false
clickable	true
enabled	true
focusable	true
focused	false
scrollable	false
long-clickable	false
password	false
selected	false

Figure 23: The first view which requires interaction in Clash of Clans

Now there are no more views which can be accessed with native object recognition so it is time to use image recognition.

Here is an example of the file MAuto creates to save the click coordinates, screenshots and query images.

```
741;293;screenshot-1.png;cropped-1.png
660;259;screenshot-2.png;cropped-2.png
894;248;screenshot-3.png;cropped-3.png
326;541;screenshot-4.png;cropped-4.png
498;463;screenshot-5.png;cropped-5.png
179;383;screenshot-6.png;cropped-6.png
395;401;screenshot-7.png;cropped-7.png
720;535;screenshot-8.png;cropped-8.png
1196;645;screenshot-9.png;cropped-9.png
192;340;screenshot-10.png;cropped-10.png
567;234;screenshot-11.png;cropped-11.png
631;643;screenshot-12.png;cropped-12.png
723;584;screenshot-13.png;cropped-13.png
468;406;screenshot-14.png;cropped-14.png
474;410;screenshot-15.png;cropped-15.png
472;457;screenshot-16.png;cropped-16.png
364;315;screenshot-17.png;cropped-17.png
164;185;screenshot-18.png;cropped-18.png
164;185;screenshot-19.png;cropped-19.png
164;185;screenshot-20.png;cropped-20.png
164;185;screenshot-21.png;cropped-21.png
638;642;screenshot-22.png;cropped-22.png
```

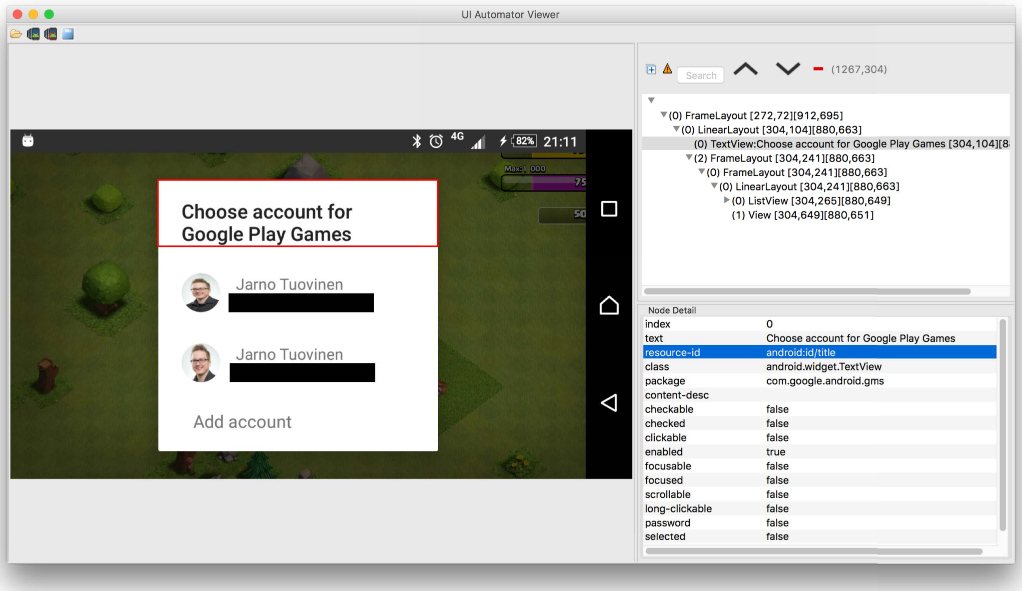
Here is an example of the Appium test script which can replay the test on every device Appium supports.

```
# -*- coding: utf-8 -*-

import os
import sys
import time
from appium import webdriver
from time import sleep
from akazeglue import *

def log(msg):
        print (time.strftime("%H:%M:%S") + ": " + msg)

def wait(image, interval=5, rounds=10):
    global akaze
    if not akaze.wait(image, interval=interval, rounds=rounds):
        sys.exit(1)
```



Choose account for Google Play Games

Jarno Tuovinen [redacted]

Jarno Tuovinen [redacted]

Add account

```
▼(0) FrameLayout [272,72][912,695]
  ▼(0) LinearLayout [304,104][880,663]
      (0) TextView:Choose account for Google Play Games [304,104][8
    ▼(2) FrameLayout [304,241][880,663]
      ▼(0) FrameLayout [304,241][880,663]
        ▼(0) LinearLayout [304,241][880,663]
          ▶(0) ListView [304,265][880,649]
            (1) View [304,649][880,651]
```

Node Detail

index	0
text	Choose account for Google Play Games
resource-id	android:id/title
class	android.widget.TextView
package	com.google.android.gms
content-desc	
checkable	false
checked	false
clickable	false
enabled	true
focusable	false
focused	false
scrollable	false
long-clickable	false
password	false
selected	false

Figure 24: Google Play account selection

```
def click(image):
    global akaze
    akaze.click(image)

def wait_click(image, interval=5, rounds=100):
    global akaze
    wait(image, interval, rounds)
    click(image)

def wait_click_elem_name(name, interval=5, rounds=100):
    global driver

    logger.debug("Waiting for item name '{}'".format(name))
    current_round = 1
    while current_round <= rounds:
        try:
            elem = driver.find_element_by_name(name)
            logger.debug("Waiting ended, item found: {}".format(name))
            elem.click()
            return True
        except Exception:
            logger.debug("Still waiting for item")
            sleep(interval)
            current_round += 1
    logger.debug("Waiting ended, item NOT found")
    return False



desired_capabilities = {}
desired_capabilities['app'] = '/Users/testdroid/projects/dcode/clash-of-clans-8-332-14.apk'
desired_capabilities['platformName'] = 'Android'
desired_capabilities['deviceName'] = 'Android Phone'
desired_capabilities['appPackage'] = 'com.supercell.clashofclans'
desired_capabilities['appActivity'] = '.GameApp'
desired_capabilities['newCommandTimeout'] = 90

appium_url = "http://localhost:4723/wd/hub"
screenshot_dir = "/Users/testdroid/projects/dcode/service/modules/asdf_server/screenshots"

driver = webdriver.Remote(appium_url, desired_capabilities)

# Initialize AKAZE Glue
akaze = AkazeGlue()
akaze.initialize(driver)

# Wait a bit
# sleep(10)

# Wait for important notice
#wait("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/important-notice.png")

# Click outside the box
```

```
#akaze.click_coord(1, 1)

# Click button, id = button3, text = OK
wait_click_elem_name("OK")

# Wait and click for profile
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/spedepekka-profile-cropped.p

# Click cancel on load village
#wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-5.png")

wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-6.png")

wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-7.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-8.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-9.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-10.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-11.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-12.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-13.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-14.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-15.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-16.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-17.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-18.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-19.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-20.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-21.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped-22.png")

log("Quitting")
driver.quit()
```

6.1.1. Things we learned from the CoC use case

It is important not to take query image too close to the object.

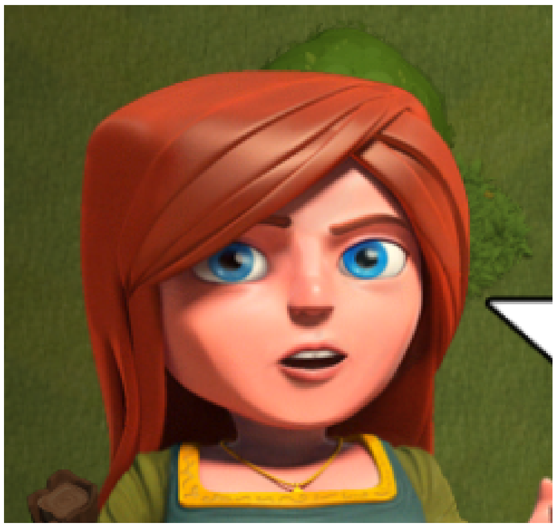


Figure 25: Good query image example.

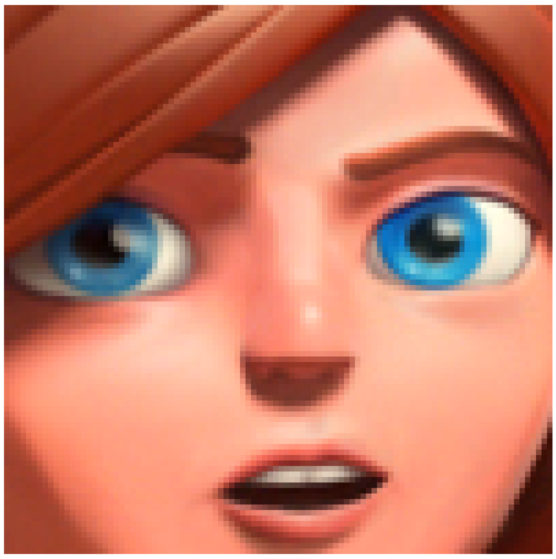


Figure 26: Bad query image example.

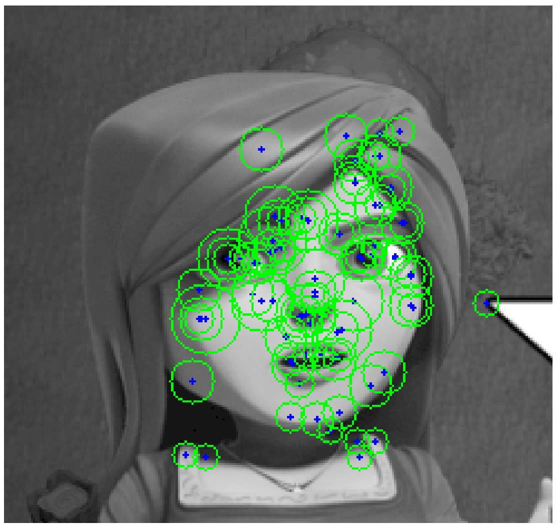


Figure 27: Found 83 AKAZE features from the good query image example.

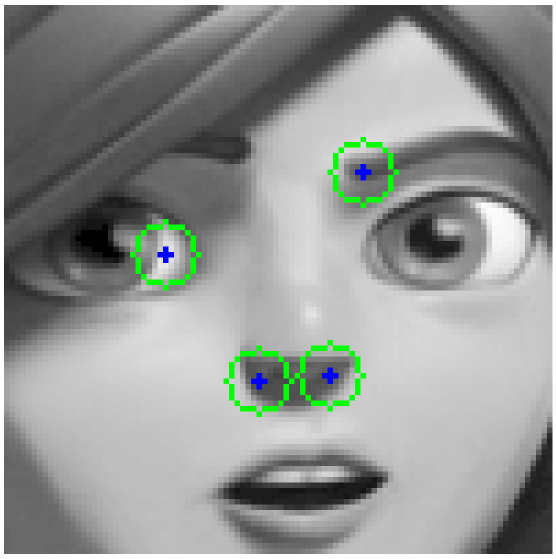


Figure 28: Found 4 AKAZE features from the bad query image example.

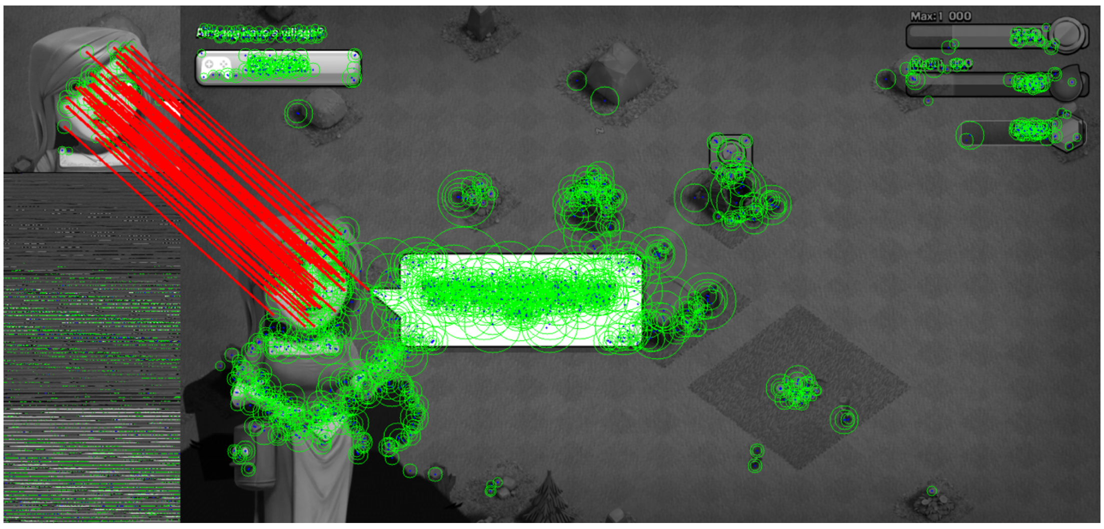


Figure 29: AKAZE can find the good query image from the screen.

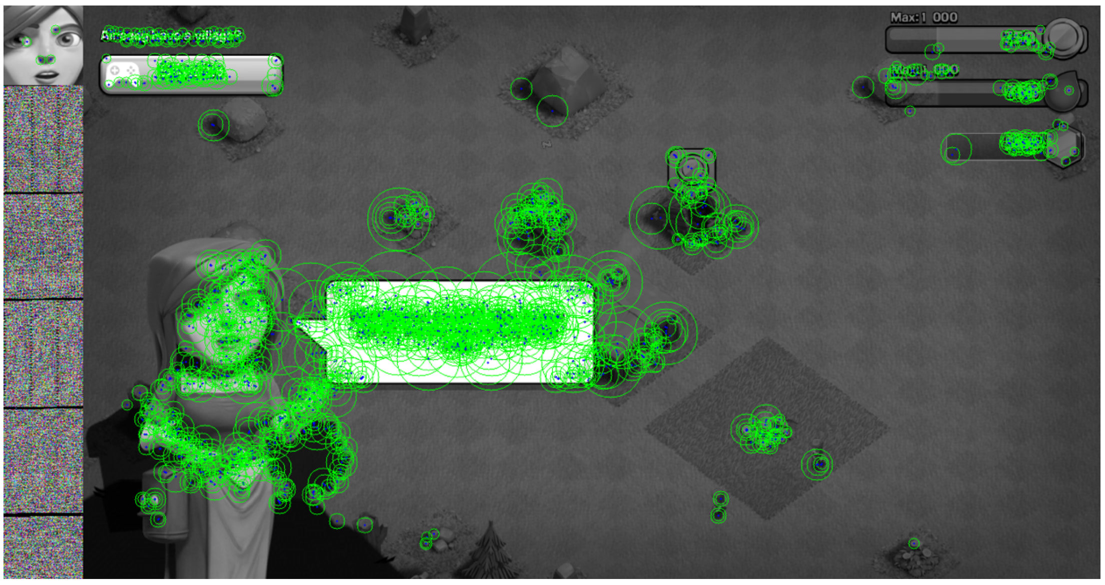


Figure 30: AKAZE cannot find the bad query image from the screen.

7. DISCUSSION

It is very time consuming for human tester to take the screenshots in the device, transfer them to host machine and crop appropriate query image. This task is very repetitive and it can be automated. MAuto is designed to do this task and decrease the manual work. It can take the screenshots and transfer the images to the host machine, but the solution to crop the query image is not the best.

We used MAuto to create automated test scripts for Clash of Clans. MAuto crops the images properly and can create reusable tests, but the user needs to have some understanding to modify the scripts to improve the performance. MAuto doesn't automate everything, but it can improve the speed of test automation script creation. The selected query images have a huge affect on test stability on other devices.

The query image must have good features to be found later from the screen. Figure 25 is a good query image, because MAuto and AKAZE found 83 features from this image (see Figure 27) and most of them can be found later from the screen as well (see Figure 29). Figure 26 is the query image from the same click, but closer, and this is not as good as the query image a bit further. MAuto and AKAZE found just 4 features (see Figure 28) and most likely this query image cannot be found from the screen when the test is run (see Figure 30).

Current version of MAuto has predefined box to crop the query image from the click coordinate and sometimes it is too small to contain usable number of features. Sometimes the user needs to manually crop better query image from the screenshot to get better results.

The user should use native object recognition whenever possible. Native object recognition is not as fragile as image based recognition. MAuto does not have the capability to understand if it would be possible to use native object recognition for the click.

When creating the tests, the screenshot operation is quite slow. It can take even couple seconds to take the screenshot. This means the usability of the application in the browser is not the same as in the device without MAuto. It is also harder to play games through the browser than in the device.

Other recording tools are far better when native object recognition can be used all the way in the application. If native object recognition cannot be used, then MAuto is the only choice. There are no alternatives at the moment for MAuto in this case.

It is not possible to give inputs to mobile device sensors with MAuto. This means it is not possible to test directly games which use sensor data. Appium has some support to sensor inputs, but MAuto cannot record those inputs.

7.1. Future work

It would be good idea to make the cropped image size dynamic. At the moment it is static 10 pixel square around the click coordinate. When cropping the query image we could calculate the number of features in the image and if there are less features than 20 for example, then the algorithm should increase the query image size and calculate the features again until the image has good amount of features. This would decrease the manual work the user has to do to fix the low quality query images.

It should be quite easy to add iOS support to MAuto as well. Appium works already for Android and iOS. The corner problems are to find a way to take a screenshot from iOS device and to find a quality VNC client for iOS. The image recognition solution works on iOS out of the box and the browser is in the host machine so it is not dependant on Android.

MAuto could recognize if the view can be accessed with native object recognition and use it when ever possible. As discussed earlier, native object recognition is far better than image recognition, but it cannot be used all the time.

MAuto cannot work with fast-paced mobile games, because it is too slow. It takes too much time to transfer the screenshot from mobile device to host. It is impossible to play fast-paced games with MAuto, because the game can end in couple seconds without new inputs.

To overcome the slowness, one solution could be to compress the image in mobile device and then send it to host machine. Other solution could be to tap into the Android operating system and remove the VNC solution completely. MAuto takes the user inputs from desktop browser and it is not ideal way to interact with mobile device. It would be better to trap the inputs directly from the screen of the mobile device and transfer the clicks and images to host machine after the test has been recorded.

If the test recording would be in the mobile device, MAuto might be able to trap the sensor inputs as well and write those inputs to tests as well.

8. CONCLUSION

The thesis focused on mobile game testing. It reviewed the motivation, key milestones and challenges. Especially, it highlighted why mobile game testing and test automation is harder than testing of traditional mobile applications. The biggest reason to this is that native object recognition doesn't work with games and the tests has to use some other object recognition methods, like image recognition.

Tools to create image based recognition test scripts doesn't exist at the moment. This thesis has introduced a testing tool, MAuto, to make it easier to create automated mobile game tests. The approach is based on image recognition. MAuto is very raw approach to solve the problems and as it is, it is not ready for end users. With some polishing MAuto would work on slow games, but it doesn't work with fast games that require rapid user interactions.

The fundamentals of image recognition has been covered in this thesis to explain why image based recognition works with mobile game testing. It is very crucial to select good query images to make the tests stable for production use.

Experts on mobile testing has answered to questionnaire and they more or less agree with the findings in this thesis. Overall expert opinion also supports the statement that mobile game testing is harder than mobile application testing. There is a need for better tools in this area. Similar tools to MAuto has not yet been released and a need for this kind of testing tool is real.

9. REFERENCES

[1] Kohl J. (2013) Tap Into Mobile Application Testing. Leanpub.

[2] Knott D. (2015) Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business. Addison-Wesley Professional, New York, 1 edition ed.

[3] Iversen J. & Eierman M. (2014) Learning Mobile App Development: A Hands-on Guide to Building Apps with IOS and Android. Pearson Education.

[4] Crispin L. & Gregory J. (2009) Agile Testing: A Practical Guide for Testers and Agile Teams. A Mike Cohn signature book, Addison Wesley Professional. URL: `https://encrypted.google.com/books?id=R2DImAEACAAJ`.

[5] Whittaker J.A. (2009) Exploratory software testing: tips, tricks, tours, and techniques to guide test design. Pearson Education.

[6] Meszaros G. (2003) Agile regression testing using record & playback , pp. 353–360URL: `http://doi.acm.org/10.1145/949344.949442`.

[7] Bach J. (1999) Test automation snake oil .

[8] Jovic M., Adamoli A., Zaparanuks D. & Hauswirth M. (2010) Automating Performance Testing of Interactive Java Applications. In: Proceedings of the 5th Workshop on Automation of Software Test, AST '10, ACM, New York, NY, USA, pp. 8–15. URL: `http://doi.acm.org/10.1145/1808266.1808268`.

[9] Alegroth E., Nass M. & Olsson H. (2013) JAutomate: A Tool for System- and Acceptance-test Automation. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 439–446.

[10] Novak J. (2008) Game Development Essentials: An Introduction. Game development essentials series, Thomson/Delmar Learning. URL: `https://books.google.fi/books?id=fnoLQAAACAAJ`.

[11] Collard J.F. & Burnstein I. (2002) Practical Software Testing. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[12] Parker J.R. (2011) Algorithms for image processing and computer vision. Wiley Pub.

[13] Lowe D.G. (2004) Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision 60, pp. 91–110. URL: `http://link.springer.com/article/10.1023/B%3AVISI.0000029664.99615.94`.

[14] Bay H., Ess A., Tuytelaars T. & Gool L.V. (2008) Speeded-up robust features (surf). Computer Vision and Image Understanding 110, pp. 346 – 359. URL: `http://www.sciencedirect.com/science/article/pii/`

S1077314207001555, similarity Matching in Computer Vision and Multimedia.

[15] Alcantarilla P. (2011) Vision based localization: from humanoid robots to visually impaired people. Ph.D. thesis, PhD thesis, University of Alcalá, Alcalá de Henares, Madrid, Spain.

[16] Alcantarilla P., Bartoli A. & Davison A. (2012) Kaze features. In: A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato & C. Schmid (eds.) Computer Vision – ECCV 2012, Lecture Notes in Computer Science, vol. 7577, Springer Berlin Heidelberg, pp. 214–227. URL: http://dx.doi.org/10.1007/978-3-642-33783-3_16.

[17] Alcantarilla P.F., Nuevo J. & Bartoli A. (2013) Fast explicit diffusion for accelerated features in nonlinear scale spaces. In: British Machine Vision Conf. (BMVC).

[18] Yeh T., Chang T.H. & Miller R.C. (2009) Sikuli: Using gui screenshots for search and automation. In: Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology, UIST '09, ACM, New York, NY, USA, pp. 183–192. URL: http://doi.acm.org/10.1145/1622176.1622213.

[19] Chang T.H., Yeh T. & Miller R.C. (2010) Gui testing using computer vision. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10, ACM, New York, NY, USA, pp. 1535–1544. URL: http://doi.acm.org/10.1145/1753326.1753555.