Baltic J. Modern Computing, Vol. 6 (2018), No. 1, 56-91 https://doi.org/10.22364/bjmc.2018.6.1.05

An Approach for iOS Applications' Testing

Ivans KUĻEŠOVS^{1,2}, Juris BORZOVS¹, Aigars SUSTERS², Vineta ARNICANE¹, Guntis ARNICANS¹, Kirils KEIDUNS², Janis SKUTELIS²

¹ Faculty of Computing, University of Latvia, Raina bulv. 19, LV-1586, Riga, Latvia ² SIA "C.T.Co", Meistaru 33, Valdlauči, Ķekava district, LV-1076, Latvia

ivans.kulesovs@gmail.com, juris.borzovs@lu.lv, aigars.susters@gmail.com, vineta.arnicane@lu.lv, guntis.arnicans@lu.lv, kiril.keidun@ctco.lv, jaanis.skutelis@gmail.com

Abstract. The mobile conquers the world. The need in a comprehensive and systemized multiedge testing approach is rising along with mobile apps becoming even more complex. As a leader in enterprise market, Apple iOS has been chosen as a target mobile platform for the study. The authors have investigated aspects that influence functional testing of iOS apps in particular, and mobile – in general. The study also exposes the security capabilities and risks that often are not mitigated in favor of time to market rush. Investigation and clustering of mobile UI test automation tools is performed. The capabilities and limitations of Apple UIAutomation are discovered. Solutions aimed at overcoming the limitations of out of the box UIAutomation are united in tTap framework developed by the authors. In conclusion an ideal cross-platform mobile UI test automation tool is proposed.

Keywords: mobile applications' testing, iOS, test automation.

1. Introduction

In 1975, the first theoretic foundation of testing by Goodenough and Gerhart (1975) was published. A year before that the first publication on software testing was published in Latvia by Barzdin et al. (1974), but an enriched version of it was presented in 1977. Since those times theory and practice of testing have evolved quite significantly through emergence of testing activists (Myers (1979/2004), Beizer (1990, 1995), Kaner (1999, 2001), Bach (Kaner et al., 2001), Pettichord (2007), Kaner et al. (2001), Black (2009), etc.) and under the influence of different software development approaches (waterfall, rapid application development, agile, etc.). Nowadays, testing has become a crucial part of the software development process.

The rise of mobile technology has touched upon the lives of everyone. According to the study by Research and Markets (2014), the mobile cloud market is expected to be worth US \$46,90 billion by 2019, while the research by Markets and Markets (2014) shows that heterogeneous mobile processing & computing market will be worth US \$61,70 billion by 2020. iOS from Apple is one of the most popular mobile operating systems. According to Citrix (2015), iOS holds 64%, and according to Good Technology

(2013), iOS holds even 73% market share of all enterprise mobile devices. According to the same study by Good Technology (2013), iPads hold 91,4% of enterprise tablets. That is why iOS has been chosen as a target platform for our research.

With the growth of platform abilities, applications become more complex to satisfy the increasing user needs (Crittercism, 2014). The increased complexity means that there are many aspects that should be taken into consideration when testing functional suitability, performance efficiency, compatibility, reliability, maintainability, and portability of iOS native business applications.

Enterprise workers are always more interested in information security than private users. The level of security is one of the factors why iOS has a dominant position in enterprise market (Eston, 2012), especially in Bring Your Own Device (BYOD) market space. While the operating system itself provides capabilities for secure application creation, they often are neglected in favor of time to market rush. That is why testing of functional security is a hot topic as well.

In order to reduce the time needed for regression testing, to spare more time for exploratory testing, or just to decrease the costs, tests to be automated. Tests can be automated at various levels. In terms of return on investments including the maintenance costs, the following test coverage model is thought to be the right one in the ideal world: most of the tests are automated at the unit level; the least of the tests are automated at the UI level; different types of integration tests lay somewhere in between. The session based/ exploratory manual testing ensures confidence in automated tests. The model is depicted in Fig. 1.1. (Scott, 2012).

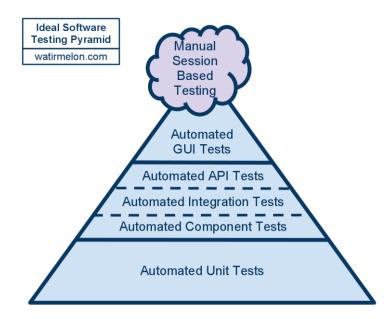


Fig. 1.1. Automated test coverage model per test level. (Scott, 2012)

While according to this model, tests at the UI level have the least coverage, these automated end-to-end tests are still very important to ensure the general confidence that the previously developed app functionality, as well as the basic UI interactions are still

up and running. Automated tests from this level are probably even more important for mobile apps, because there are many gestures like tap, double tap, swipe, drag, etc. to be checked.

Several solutions have already been created or adapted for mobile UI test automation, in particular, for iOS apps. However, they all have their pros and cons, while there is no study that exposes them to choose and adapt the right one for the environmental context of a testing organization. There are also no studies that investigate the capabilities and limitations of OEM Apple UIAutomation tool.

All this increases the need for the multi-edge iOS applications testing approach that extends the systemized knowledge in the mobile testing field in general. Solutions that overcome part of the limitations of a native automator like changing the connectivity, assertions based on image comparison, advanced UI element search, repeatable executor for checking the wait conditions, simulation of memory warnings, etc. are united under tTap extension developed by the authors.

2. Inventory of Testing Ideas and Structuring of Testing Terms

2.1. Inventory of Testing Ideas

The authors have performed the inventory of testing ideas (Kuļešovs et al., 2013). It was inspired by Kalninsh and Borzovs (1981). This activity resulted into the ideas division among the following eight classes:

- Fundamental ideas.
- How to detect the correctness of the test result?
- How to detect the completeness of the testing?
- How to test (approach, method, technique)?
- What to test (object)?
- Which quality attribute (characteristic) to test?
- When to test (phase)?
- Unclassified.

Three millennial fundamental testing ideas are:

• Errare humanum est – To err is human.

• Aliena vitia in oculis habemus, a tergo nostra sunt - The vices of others we have in the eyes, in the rear of our own.

• In propria causa nemo judex - No one can be judge in his own cause.

Other testing ideas were identified through analyzing the testing terms from ISTQB Glossary (ISTQB, 2012). As a result, a map showing the linkage between the testing relation the definite terms and their to class was generated (see http://science.df.lu.lv/kaab13). It was produced using the tool that adopts the term graph building algorithm developed by Arnicans, Romans, and Straujums (Arnicans et al., 2013), (Arnicans and Straujums, 2012).

2.2. Software Testing Review on Meta-level

From practical point of view software testing mainly can be expressed by *testing strategy* and *testing tactics* on the meta-level (i.e. on the higher level of abstraction). Contexts of real testing project and theoretical background and experience of testing

58

team influence the selection of the strategy and/ or tactics and the usage of principles of testing schools in the current testing project or campaign. A software testing review on meta-level is depicted in Fig. 2.1.

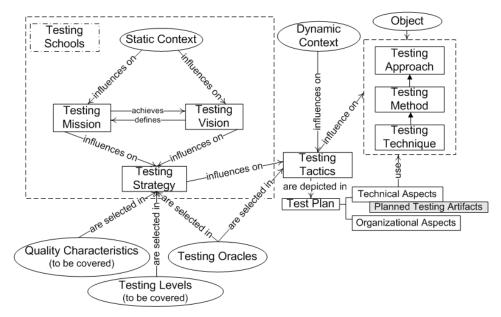


Fig. 2.1. Software testing review on meta-level.

Static context influences very much the testing vision and testing mission. Static context depends on the type of the organization (i.e. governmental, outsourcer, start-up, etc.) and on the type of the software produced (enterprise software, commercial software, web page, etc.). The options mentioned above are generally static during the whole product lifecycle. Testing vision denotes the aims that the testing team wants to achieve by testing. In some cases testing vision can be focused on producing the software with all high and critical software failures discovered and fixed, and 95% of medium severity failures identified. In some cases it can be to receive an acceptance sign-off of the product from the customer. Testing mission denotes actions that testing team does in order to achieve the testing vision. For example, the team can use only scripted testing, or it can use the benefits of the exploratory testing as well, to receive an acceptance sign-off of the product from the customer. Or testers prepare automated tests before the development to keep the product always deliverable to the customer as test-driven development suggests. Testing schools are theoretical frameworks that define testing vision and testing mission based on the static context.

All aspects of testing schools (it can also be the mix of aspects from different schools) that prevail within the organization and are common for the definite product type influence the testing strategy of the given software project. *Testing strategy* describes a general approach for testing. *Testing strategy* consists of the specification of the roles and responsibilities of each person involved in testing, testing levels, environment requirements, overall testing schedule, testing tools, risks and its mitigations, testing priorities, testing status reporting, etc.

Testing oracles that define testing exit-criteria and those that are used as the source of the derivation of test cases and expected results (i.e. correctness oracles) should be chosen within the testing strategy definition. The selection of *quality characteristics* to be covered by testing process should occur during the definition of testing strategy as well. Test results completeness oracles can be defined when selecting testing tactics, because often there are much more details about expected results amount available during tactics selection process.

Dynamic context depends on the project phase and influences the choice of the testing tactics that are appropriate for the given time frame, for the definite object under test, and for the current micro testing goal. Examples of the dynamic context factors are fulfillment of test entry criteria in time, availability of shared testing resources, the stabilization and bug fixing phase of the development process, etc. *Testing tactics* should be consistent with the testing strategy.

Testing tactics for each object under test are depicted in the *test plan*. Test plan consists of organizational and technical aspects. Testing tactic also influences the choice of the *testing approach* to be used to fulfill the current micro testing goals. Thus, technical aspects of the test plan should include the selection of the appropriate testing approaches, methods, and techniques. Testing artifacts (like test cases, test suites, traceability matrix, test data, etc.) to be produced by the testing process should be mentioned in the test plan as well. It is worth noting that some schools do not require formal and written test plans as a mandatory artifact of testing process.

2.3. Testing Schools

Testing society distinguishes five testing schools (Pettichord, 2009). They are:

- Analytic School;
- Standard School;
- Quality School;
- Context-Driven School;
- Agile School.

The schools are frameworks for categorization of test engineers' believes about testing and are their guide on the testing process. Testing schools are not competitive; they can be used in the collaborative mode as well. They all have exemplar techniques or paradigms, but they are not limited to them.

During the last decade the agile school starts taking the dominant position. The paradigm of this school is that testing process proves that system under test or its particular functionality work as expected, while the focus of other schools is to find the errors. Test-driven development is one of the agile testing school paradigms, thus automated acceptance tests are demonstrative exemplar of the school.

2.4. Systematization of Testing Terms: Approach, Method, and Technique

Despite the attempts of standardization of testing terms and ideas by different authorities, such as ISTQB and IEEE, there is still a little chaos prevailing in the testing literature, and between the testers themselves on the explicit usage and definition of the terms.

The connection and clear border between *testing approach*, *testing method*, and *testing technique* are not defined in the testing theory. For example, Beizer (1995)

defines test technique as a systematic method: "A test strategy or test technique is a systematic method used to select and/or generate tests to be included in a test suite." In the same time, he uses test technique and test method as completely equal statements: "... here I present you with ready-made equivalence class partitioning methods (or test techniques) ..." (Beizer, 1995); "[T]est execution technique: The method used to perform the actual test execution, either manual or automated" (ISTQB, 2012). Other authors, such as Kaner et al. (1999, 2001), Pressman (2005), and Sommerville (2007) have a mix of using words technique, method, approach, and strategy in regard to testing as well.

The attempts of making a distinction between approach, method, and technique were already performed by language teaching specialists in 1963, 12 years before the first theoretic foundation of testing by Goodenough and Gerhart was published. In 1963 Anthony provided "much needed coherence to the conception and representation of elements that constitute language teaching:" (as cited in Kumaravadivelu (2006))

An *approach* is "a set of correlative assumptions dealing with the nature of language and the nature of language teaching and learning. It describes the nature of the subject matter to be taught. It states a point of view, a philosophy, an article faith..."

A *method* is "an overall plan for the orderly presentation of language material, no part of which contradicts, and all of which is based on the selected approach. An approach is axiomatic, a method is procedural".

A *technique* is described as "a particular trick, stratagem, or contrivance used to accomplish an immediate objective".

"The arrangement is hierarchical. The organizational key is that techniques carry out a method which is consistent with an approach."

In 1982 Richards and Rogers (as cited in Kumaravadivelu (2006)) performed an attempt to enhance the framework developed by Anthony through dividing language teaching process into approach, design, and procedure. But, despite rather vague definition of terms *approach*, *method*, and *technique*, and not considering in any way of complex connections between them, exactly these terms are in favor of the most current teacher training manuals. (Hall, 2011)

We suggest systemizing testing approach, testing method, and testing technique in the same hierarchical way, using the experience and keeping in mind the mistakes of language teaching specialist. Schematic relation between terms mentioned above is shown in Fig. 2.2.

Approach	
Method 1 Method 1 Method N Tashsigus 1 Tashsigus 2 Tashsigus M 2 Tashsigus M 1 Tas	
Technique 1 Technique 2 Technique 3 Technique M-2 Technique M-1 Technique M-1	nique M

Fig. 2.2. Relation between approach, method, and technique.

Testing approach "states a point of view, a philosophy, an article faith" that a test engineer takes when designing test cases.

Testing method is "an overall plan for the orderly presentation" of testing techniques.

Testing technique is "a particular trick, stratagem, or contrivance" to design a test case.

Testing techniques are united under testing methods based on the test case design formality (for black-box testing approach) or based on other common pronounced attributes (for white-box testing approach).

The "organizational key" stays the same as suggested by Anthony – "techniques carry out a method which is consistent with an approach".

2.5. Black-box Testing

Black-box is a software testing approach when test engineer designs test cases as if she does not know anything about the internal structure of the software under test.

Black-box testing approach consists of seven testing methods that are differentiated based on the source used for test case design process and based on the level of formality of test case designs. The relation between black-box testing methods and techniques is shown in Fig. 2.3.

Approach Memods Model-based User Story Defect-based Experience-based Random Syntax Specification- based Testing Model-based User Story Defect-based Testing Fuzz Testing Testing Fuzz Testing Testing Testing Testing Testing Testing Fuzz Testing T			-	lack-box			
Specification-based Testing Model-based Testing User Story Testing Defect-based Testing Experience-based Testing Random Testing Syntax Testing Domain Analysis State Transition Testing Defect Taxonomies Checklist-based Testing Fuzz Testing Equivalence Class State Transition Diagram Type Testing Experiatory Testing Boundary Value Use Case Testing Use Case Testing Environment Error Guessing Logic-based Testing Decision Table Decision Table Failure Symptoms Ad-hoc Testing				pproacn			
Domain Analysis State Transition Testing Defect Taxonomies Checklist-based Fuzz Testing Equivalence Class State Transition Diagram Type Testing Exploratory Testing Partitioning State Transition Table Root Cause Exploratory Testing Error Guessing Boundary Value Use Case Testing Failure Symptoms Ad-hoc Testing Logic-based Testing Decision Table Exploratory Testing	Specification-						
Combinatorial Testing	Equivalence Class Partitioning Boundary Value Analysis Logic-based Testing Decision Table KV Chart	State Tra State Ti State Use C	nsition Testing ransition Diagram Transition Table	Roo	Type t Cause	Tes Explorato Error G	ting ry Testing uessing
State Transition Diagram State Transition Table Use Case Testing							

Fig. 2.3. Black-box Approach

2.5.1. White-box Testing

White-box is a software testing approach when test engineer designs test cases based on the internal structure of the software under test. There are three most known white box testing methods: control flow testing, data flow testing, and mutation testing. The relation between the white-box testing methods and techniques is shown in Fig. 2.4.

2.5.2. In-Operational Testing

Classical *black-box* and *white-box* testing approaches are mainly used applied during software development. Nowadays the new *in-operational testing* approach emerges. It means that validation occurs during the system run-time. Execution environment testing (Rauhvarger and Bicevskis, 2008), self-testing (Diebelis and Bicevskis, 2008), runtime verification of business process execution (Oditis and Bicevskis, 2015a,b) are the methods united under this approach. These testing methods are part of so called smart technologies idea that software should behave as a living being and adapt to, optimize in

and defend itself again changing environment (Bicevska et al., 2015; Bicevskis et al., 2016). The relation between the *in-operational* testing methods and techniques is shown in Fig. 2.5.

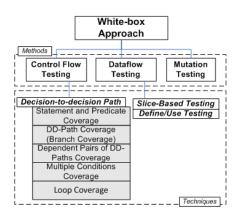


Fig. 2.4. White-box Approach

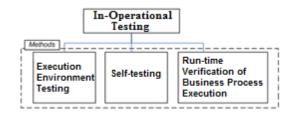


Fig. 2.5. In-Operational Testing

3. Apple iOS Applications Testing Aspects

3.1. Introduction

According to the different studies (Research and Markets, 2014; Markets and Markets, 2014; Citrix, 2015; Good Technology, 2013) iOS devices hold the major market share among the corporate workers.

With the growth of platform abilities applications become more complex (Crittercism, 2014) to satisfy the increasing user needs. The increased complexity means that there are many aspects that should be taken into consideration when testing mobile applications. Mobile workers mostly use native business applications on their devices; otherwise there would not be such a dominant position of the single operating system. That is why iOS native applications are the subject of the main interest for this section and study in general.

Despite the fact that the topic being hot, there are only some academic studies (Muccini et al., 2012; Dantas et al., 2009; Gao et al., 2014) performed that systemize the generic aspects that should be taken into consideration when testing the mobile applications without specifying the platform. Other studies – by Franke et al. (2011,

2012) that include the clear distinction between the platforms, concentrate on some narrow topic. On the other side, there are different iOS testing checklists, mind maps, blogs, etc. available in the internet. This motivates the authors to perform the systematic literature review of academic literature in the field of mobile testing and perform the literature review of the available non-academic (or multivocal, as per Ogawa and Malen, 1991) sources in the field of iOS testing (Kulesovs, 2015).

It was decided to concentrate both reviews on aspects of manual testing of such quality characteristics as functional suitability, performance efficiency, compatibility, reliability, maintainability, and portability according to ISO (2011). Usability testing is out of scope (except parts that are closely related to or are on the border line with the quality characteristics mentioned above).

The following research question was formulated:

RQ: Which aspects (i.e. features and/ or limitations) influence the testing of functional suitability, performance efficiency, compatibility, reliability, maintainability, and portability of the iOS native business applications?

The results of both reviews are merged in order to answer the research question.

3.2. Research Methodology

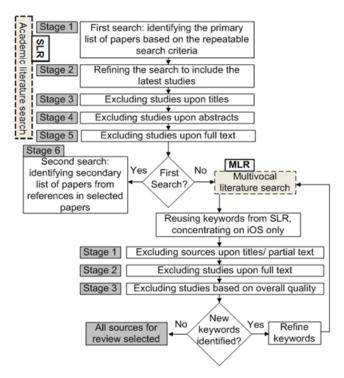


Fig. 3.1. Process of sources selection for SLR and MLR.

The systematic literature review (SLR) of the academic sources was performed in order to gain the aspects of the mobile applications testing. The multivocal literature

review (MLR) was performed in order to gain the exclusive aspects of iOS applications testing. The idea to perform two types of the review to consolidate the data from different sources was taken from work by Tom et al (2013). Fig. 3.1 shows the stages of sources selection for the whole review process applied in this study.

The procedure described by Kitchenham and Charters (2007) was followed in order to conduct the systematic literature review. The qualitative review approach was applied in order to include a rigor into the systematic review of multivocal literature as suggested by Ogawa and Malen (1991). They define the multivocal sources as accessible, but non-academic writings on the topic.

3.3. Results

3.3.1. Summary of Reviews

Despite the fact that the search criteria for SLR includes studies starting from 2007, the first selected study by Dantas et al. (2009) was published in 2009, but the most productive years are 2012 (five studies: Franke et al. (2012a, 2012b), Kim (2012), Marinho and Resende (2012), and Muccini et al. (2012)) and 2013 (three studies: Amalfitano et al. (2013), Haller (2013), and Khalid (2013)). Two studies by Franke et al. (2011) and by Franke and Weise (2011) were published in 2011, and one study by Gao et al. (2014) was published in 2014. Two studies by Franke et al. (2012a, 2011) are related to narrow topic of mobile application lifecycle, one study by Khalid (2013) is related to user complaints about iOS applications, and other nine sources by Dantas et al. (2009), Franke et al. (2012b), Kim (2012), Marinho and Resende (2012), Muccini et al. (2012), Amalfitano et al. (2013), Haller (2013), Franke and Weise (2011), and Gao et al. (2014) are related to the general testing of mobile applications.

Between the sources selected through MLR, seven sources by App Quality Alliance (2013), Pound (2013), Nearsoft (2013), TestElf (2013), uTest (2013), Neglected Potential (2013), and Addey (2013) were published in 2013, and one source was published in 2012 by Land (2012) and one in 2014 by SmartBear (2014). Five sources by App Quality Alliance (2013), Nearsoft (2013), TestElf (2013), Land (2012), SmartBear (2014) are blog posts, two sources by App Quality Alliance (2013) and Addey (2013) are testing checklists, one source by uTest (2013) is a white paper, and one source by Neglected Potential (2013) is a mind map. All the blog posts describe the testing only of one or some aspects, while other sources try to cover the whole iOS testing field.

3.3.2. Aspects of iOS Applications Testing

The aspects that influence the testing of iOS applications gathered through SLR and MLR are shown in Table 3.1. If a source is referred in the table before the details of an aspect, it means that aspect is just mentioned in the source without pointing the details that are related to iOS applications testing.

There are three types of iOS devices: iPad, iPhone, and iPod mentioned in App Quality Alliance (2013), uTest (2013), Neglected Potential (2013), and Addey (2013) that have different screen size, resolution and pixel ratio, processing efficiency, memory, and storage capacity, as per Dantas et al. (2009), Kim (2012), Marinho and Resende (2012), Muccini et al. (2012), Haller (2013), Khalid (2013), Franke and Weise (2011),

Table 3.1 Aspects of iOS Applications Testing

		Envi	conment
Hardware			
Devices	iPad, iPl Apple W	none, iPod Vatch.	Screen size, resolution & pixel ratio, pro-cesssing efficiency, memory, storage capacity
Simulator			
External	Headpho	ones, keyboar	d; wired/ wireless.
Accessories			
Operating System	0.0	-	
OS Variety	OS upgr	ade.	
Restrictions and			AP (in-app purchase), Location Services,
Privacy Settings			Photos, Social Networking, Microphone,
_	Motion	Activities, Ce	llular Data Use, Background App Refresh.
Resources			
Limitations		storage, amou ng capabilitie	nnt of memory, running out of battery, s.
Consumption	Memory	consumption	, battery consumption.
Connectivity			
Network Types			ks, Bluetooth, Airplane mode.
Network	-	-	ection; connection loss.
Conditions	Ask for	connection.	
Internalization	Dí	<u>, 1</u>	
Region Formats Date/ Time		mat, hour for	
Settings	Switchin	ig between ti	ne zones, system time too fast/ too slow.
		Applicati	on Lifecycle
Installing and Lau	nching		
Background			
Crash Low-Memory War	minaa		
_	mings	C 11/ CMC	
Interruptions		audio/ video	push notifications, system alerts, GPS signal,
Application Updat	te		
		Inside the	Application
Keyboard		Extended ke	•
Data Import/ Exp		Email; Blue	tooth/ network (peer to peer).
Logging/ Analytics	5		
In-App Purchases			
Web View			
		U	I/ UX
Gestures			
Smooth Animation	1		
Pull to Refresh Orientation		Dortroit 1	dsaana
Half Pixels		Portrait, lan	uscape.
Localization		Native chara	acters and special symbols.
Accessibility			accessibility zoom, etc.
Accessionity		voiceOver,	accessionity 200m, etc.

66

Gao et al. (2014), uTest (2013), Neglected Potential (2013), and Addey (2013). It is claimed in Dantas et al. (2009) that functionalities, usability issues in the interface design, and user behavior "to be tested in emulator", while other sources Kim (2012), Haller (2013), Khalid (2013), Franke and Weise (2011), and SmartBear (2014) state that almost everything should be tested on the real device to get the reliable test results.

There are also different types of the external accessories, both wired and wireless (Franke et al., 2012b), (Amalfitano et al., 2013) like headphones (Franke et al., 2012b), (Amalfitano et al., 2013), (Addey, 2013) and keyboard (Franke et al., 2012b), (Amalfitano et al., 2013) that can be connected to the device.

It is claimed in many sources like in Kim (2012), Muccini et al. (2012), Haller (2013), Gao et al. (2014), uTest (2013), Neglected Potential (2013), and Addey (2013) that the variety of operating systems (OS) is an important testing aspect, while OS upgrade is mentioned explicitly only in Haller (2013). It is possible to set the restrictions on the usage of different hardware or OEM software completely or for the specific application within the iOS (App Quality Alliance, 2013), (TestElf, 2013), (Neglected Potential, 2013).

Mobile devices have limited power, processing, and memory resource (Dantas et al., 2009), (Franke et al., 2012a, b), (Marinho and Resende, 2012), (Muccini et al., 2012), (Haller, 2013), (Khalid, 2013). Thus resources consumption efficiency plays an important role in application success (Dantas et al., 2009), (Franke et al., 2012b), (Marinho and Resende, 2012), (Muccini et al., 2012), (Khalid, 2013), (App Quality Alliance, 2013). Applications should also be checked on different networks, i.e. strong WiFi connection, cellular network (LTE, 3G, EDGE), and in Airplane mode (Dantas et al., 2009), (Franke et al., 2012b), (Muccini et al., 2012), (Gao et al., 2014), (Nearsoft, 2013), (TestElf, 2013), (Neglected Potential, 2013), (Addey, 2013). Different network conditions (e.g. slow connection, packets loss, etc.) should be taken into consideration as well (Neglected Potential, 2013). Different regional settings, like data and time formats (Neglected Potential, 2013), (Addey, 2013), as well as time zones (Addey, 2013) are also the subject of interest.

iOS application lifecycle consists of several phases, and there are specific conditions that can uniquely influence application's behavior while being in the definite phase. An application can be just installed and launched for the first time (Marinho and Resende, 2012), (App Quality Alliance, 2013), (Addey, 2013), work in foreground, stay in background (Franke et al., 2012a,b, 2011), (App Quality Alliance, 2013), (Neglected Potential, 2013), receive memory warnings (Franke et al., 2012a,b, 2011), (Franke and Weise, 2011), (Addey, 2013), be interrupted by a call or SMS (Amalfitano et al., 2013), (App Quality Alliance, 2013), system alert (Amalfitano et al., 2013), push notification (App Quality Alliance, 2013), (Neglected Potential, 2013), GPS signal (Amalfitano et al., 2013), or audio/ video from another application (App Quality Alliance, 2013), (Neglected Potential, 2013), (IT est, 2013), (Neglected Potential, 2013). Or it can also be updated to the next version (Haller, 2013), (Neglected Potential, 2013), (Addey, 2013).

TestElf (2013) warns about the need to check an extended (Asian) on-screen keyboard, while Dantas et al. (2009) mentions on-screen keyboard as a generic aspect that should be taken into consideration. According to App Quality Alliance (2013) and Neglected Potential (2013) data can be shared via email or Bluetooth, or another network between the applications. According to Neglected Potential (2013) and Addey (2013) it is necessary to check application's logging and analytics features. Testing of In-App

Purchase component is mentioned in Neglected Potential (2013). Testing of Web View component is mentioned both in Gao et al. (2014) and Neglected Potential (2013).

An application can be manipulated with a variety of gestures (Gao et al., 2014), (Neglected Potential, 2013). When animated transitions occur, they must run smoothly (Khalid, 2013), (Neglected Potential, 2013) irrespectively of the task executed in parallel. Testing for half pixels glitches and testing of Pull to Refresh feature are mentioned in Neglected Potential (2013). The necessity of checking the application both in portrait and landscape is noticed in Dantas et al. (2009), Franke et al. (2012b), Neglected Potential (2013), and Addey (2013). The importance of localization testing is mentioned in Haller (2013) and Addey (2013). App Quality Alliance (2013) identifies the need for testing of native characters and special symbols. It should also be checked that application works as designed when accessibility features of OS are enabled (Pound, 2013), (uTest, 2013), (Neglected Potential, 2013), (Addey, 2013), (Land, 2012).

3.4. Conclusions

To conclude, straight functional testing of mobile apps cannot differ from testing of web or desktop applications. The difference mainly occurs in the aspects related to the environmental multeity.

Variety of operation systems, hardware, operating system versions (often completely rebuilt from scratch) and modifications, screen sizes, screen resolutions and densities, browsers and their versions, makes mobile as a separate universe. This makes test team to select the most covering representatives from the universe to perform testing on. In many cases there are more representatives to be selected for cross-platform mobile app testing then it is needed for desktop or web application testing.

Mobile apps usually combine desktop and web applications behavior because they need to be available both online and offline. They also have more lifecycle states to verify in comparison to desktop or web applications can have. Online nature of many mobile apps implies checking their behavior under various network conditions.

Quite limited storage, battery, and processing power resources of the mobile devices leads to the additional checks to be performed when testing mobile apps under resources shortage conditions. There are also additional checks to be performed in order to verify that available resources are used efficiently by the app.

The single smaller touch screen interaction mechanism of mobile devices forces to check the functional usability of mobile app screens. Improper positioning of UI elements on the screen and unexpected gesture interference can lead to inability to use the in-app features. The examples could be: elements position is linked to on-screen keyboard position while only the default keyboard position is taken into consideration during the design; in-app near the screen border swipes interfere with system near the screen border swipes that open various system popovers instead. The smaller screen size and interaction capabilities of mobile device extract mobile web app testing into separate topic in comparison to the standard web application testing.

4. Apple iOS Applications Functional Security Testing

4.1. Introduction

While iOS offers the variety of security enhancement features to be used within the apps, they are often neglected in favor of time to market rush. The security basics that should be tested by test specialist are: using of secure network protocols, encryption of data base, and denying the access to application data when device is locked with passcode. One of the advanced functional security testing items is a checking of the development settings file (plist) entries in production app version. (Kulešovs, 2017)

4.2. Usage of Secure Network Protocols

Usage of secure network protocols (HTTPS – HTTP over SSL or HTTP over TLS, etc.) can be ensured by intercepting the network traffic with the apps like Charles¹, Fiddler², etc. It is also possible to see the encrypted content exchange as a plain data between the mobile app and the backend with such kind of tools, i.e. to perform manual integration testing.

4.3. Data Base Encryption

This is quite a basic, but very important task for test specialist to ensure that data base is encrypted in productive app version, because it is often kept unencrypted for testing purposes during development. It is possible to download the application data base directly through Xcode³, or using such third party apps like iFunBox⁴ or iPhoneExplorer⁵. Then it is verified if encrypted or not by opening data base by any SQLite data base viewer. This also allows verifying some functional corner cases or data base corruption cases, because data base can be changed and uploaded back to iPad using the same stack of tools.

4.4. Locking the Application Data

iOS allows locking the access to application data on the device locked with passcode. However, this should be managed by app itself. That is why this feature should be often rechecked, because data to lock should be explicitly defined. During new functionality development this part is often forgotten. In-memory decryption could be another possible requirement for the high-risk apps, otherwise, currently used data is being kept in unencrypted way in Cache folder. Even if there is a code block that tries to remove all unencrypted data after usage, there is a chance that it will be left unencrypted upon app crash or app removing from memory if another app needs more memory during its execution.

¹ <u>http://www.charlesproxy.com/</u>

² <u>http://www.telerik.com/fiddler</u>

³ https://developer.apple.com/xcode/

⁴ <u>http://www.i-funbox.com/</u>

⁵ https://www.macroplant.com/iexplorer/

4.5. Advanced Functional Security Testing

One of the advanced functional security testing items is a checking of the existence of development settings file entries (NSUserDefaults) in production app version. The development settings entries could be: skipping login, using unencrypted database, choosing the advanced subscription, usage of a feature that should be bought using inapp purchase, unhiding the features currently under development, advanced debugging, and all other staff that needs some extensive interaction with app to be achieved. There are different cases how attackers can learn about those development features:

• Development and test settings files could be left together with production settings file, but not used (see Fig. 4.1).

• Entry points for development and test settings are left accessible within run-time of production version.

				-
My Ma	c	Name	Size	Туре
▼ 📲	Ivan's iPad(iPad Mini 3	com.myorg.myapp.dev.plist	213 B	property list
	A User Applications	com.myorg.myapp.plist	213 B	property list
	A Cabinet	📄 com.myorg.myapp.train.plist	213 B	property list
	AGarageBand			
	AiMovie			
	A iTunes U			
	A MobileIron			
	A			
	A Numbers			
	A Pages			
•	🯓 App File Sharing			
	2 Shortcuts			
	General Storage			
	Camera			
	iBooks			
	Voice Memos	(What was a surface to be a s		
	Raw File System	Library Preference	s	
	V haw the System		0	

Fig. 4.1. Example of the development and test settings files in production build.

This means that the first action item should be the adjusting of build process to leave only production settings file for the production build. The second action item should be, respectively, profiling the application code to disable the entry points for development and test settings for production build.

If not, then app settings file could be accessed using the apps mentioned before (Xcode, iFunBox, iPhoneExplorer), modified, i.e. appropriate development or test settings could be added, file could be uploaded back to iPad, and attacker could enjoy the benefits.

If the first action item is a self-explanatory, then to understand the severity of the second action item some more information on how to break the app without accessing the development and test settings files should be given.

70

It is possible to get the run-time properties of the app on the jail-broken device using such apps available on Cydia App Store like Cycript⁶, iNalyzer⁷, etc. These properties are shown in key-value format, even if they are not set from the current settings file. Then attacker just adds the desired settings and their values into the settings file and uploads it back to iPad to enjoy the benefits.

4.5.1. Discussion and Implications

While application security in most cases is tested by the security specialists, it is cheaper to verify that security mechanisms provided by OS vendor are used as much as possible (if the nature of the app needs it, of course) before giving the app to them. The suggestions given above allow decreasing the panic when app is checked for security when it is already in or close to production. From the authors' experience, it is often the case when product owners rush to release the app, while outsourced security specialist overloaded schedule does not allow performing the check before the target date.

5. Mobile Applications UI Test Automation

5.1. Introduction

As already mentioned in the introductory part of the thesis, in order to reduce the time needed for the regression testing and to make more time available for the exploratory testing or just to decrease the costs tests tend to be automated.

Tests could be automated in the various levels. In terms of return on investments including the maintenance costs the test coverage model depicted in Fig. 1 is thought to be the right one in the ideal world: the most of the tests are automated on the unit level; the least of the tests are automated on the UI level; different types of the integration tests lay somewhere in between. The session based/ exploratory manual testing ensures confidence in automated tests. (Scott, 2012)

While according to this model the tests on UI level have the least coverage, these automated end to end tests are still very important to give the general confidence that previously developed app functionality, as well as basic UI interactions are still up and running. Automated tests from this level are probably even more important for the mobile apps because there are many gestures like tap, double tap, swipe, drag, etc. to be checked.

5.2. Solutions for Automated UI Testing of Mobile Apps

There are several solutions already created/ adapted for mobile UI test automation, in particular, for iOS apps. The solutions could be divided into several groups based on the origin, cross-platformance, and the way of executing the automated commands.

The first big clusters are OEM automation tools vs. the third party automation tools. OEM automation tools come together with the mobile OS manufacturer IDE. All other mobile automation tools are the 3rd party solutions. The most of the solutions use API-

⁶ <u>http://www.cycript.org/</u>

⁷ https://appsec-labs.com/inalyzer/

based approach for recognizing the object on the screen, while there are some solutions that use image-based approach for the same purpose. API-based solutions can be divided into two more groups: wrappers above the native automation tools vs. others that have the prerequisite to incorporate the custom library into the app source code. Some of the solutions offer to run the tests in cloud. While almost each solution nowadays can run tests both on device and on simulator on premises, only some solutions support running the tests on the real devices in cloud.

5.2.1. OEM Automation Tools

5.2.1.1. Apple UIAutomation and XCTest

UIAutomation tests are written in JavaScript. The framework consists of the most basic functions for all UI elements available in iOS. (Apple Developer, 2012) The access to some device functions like sending app to background, changing the volume, setting the location, etc. is also available. If some custom UI View is used inside the app it can be accessed as UIAElement class – the superclass for all user interface elements in the context of the UIAutomation.

Starting from XCode 7 Apple added the possibility of writing the UI automated tests on Swift language and to run them on XCTest framework (it is a unit test style framework for Swift/ Objective-C code) inside Xcode IDE itself. In terms of the functional scope of API both UIAutomation and XCTest frameworks are on the same level.

Being the frameworks with the powerful set of basic functions, one of the issues for both of them is that the commonly used test notations from these basic functions are quite wordy. Several extensions have been created for JavaScript based UIAutomation in order to enable the ability to write the tests using the less repetitive higher level commands in a style more common for the testers. Each extension follows the notation style convenient for the creator. Both most popular extensions are distributed under MIT license. There are no extensions available for XCTest UI testing framework yet.

Tuneup JS

The main achievement of TuneupJS⁸ is the creation of the unit test like test runner and providing the extensive set of assertions. The extension has the image comparator inside that is based on ImageMagic⁹ tool. It also consists from the set of the commands that combine several UIAutomation basic commands into one higher level command making the notation shorter.

mechanic.js

mechanic.js¹⁰ is a CSS-style selector engine for UIAutomation. It also allows accessing UIAElements and executing the commands with a shorter notation.

72

⁸ <u>http://www.tuneupjs.org/</u>

⁹ http://www.imagemagick.org/

¹⁰ http://www.cozykozy.com/mechanicis/

5.2.1.1. Google Testing Support Library

Google Testing Support Library consists of three main parts¹¹:

• AndroidJUnitRunner: JUnit 4-compatible test runner for Android.

• Espresso: UI testing framework; suitable for functional UI testing within an app.

• UI Automator: UI testing framework; suitable for cross-app functional UI testing across system and installed apps.

UI Automator functionality on Android is similar to UIAutomation functionality on iOS, while Espresso could be described as white-box UI test automation tool. Testing Support Library Tests are written in Java.

5.2.1.2. Microsoft Coded UI Tests

Coded UI Tests is an analogue for UI test automation for Windows apps¹². This tool supports almost all Windows-based platforms, not only mobile ones. It could be even used for web apps UI test automation. The tests are written in C#.

5.2.2. 3rd Party Solutions and Summary

The market players with characteristics they posses are summarized in Table 5.1., Table 5.2, and Table 5.3. All of the solutions have record/ play capabilities. That is why this option is excluded from the comparison tables. Each solution also supports the CI setup.

Name	Scripting	Native/	Web	Cloud
	Languages	Hybrid		Support
Apple UI	JavaScript,	Х	+/-	-
Automation/	Swift		(need to wrap the	
XCTest			website into native app)	
Google Testing	Java	Х	+/-	-
Support Library			(need to wrap the	
			website into native app)	
Microsoft Coded	C#, VB.Net	Х	X	-
UI Tests				

 Table 5.1. OEM Solutions for Mobile UI Test Automation

¹¹ http://developer.android.com/tools/testing-support-library/index.html

¹² https://msdn.microsoft.com/en-us/library/dd380742.aspx

Name	Wrap per	API- based	Image- based	3rd Party Library in Use (If Not Own)	3rd Party Library Integration into Source Code (for iOS)
Appium ¹³	X	Х		Implements Selenium WebDriver	
Xamarin Test Cloud ¹⁴		Х		Calabash	Х
Tosca Mobile+ ¹⁵		Х		Modified MonkeyTal k, Sikuli	Х
Telerik Test Studio Mobile ¹⁶		Х			Х
DeviceAnywhere ¹⁷		Х			Х
Ranorex ¹⁸		Х			Х
SeeTest ¹⁹		Х			Х
Sikuli ²⁰			Х		Х
EggPlant ²¹			Х		Х

Table 5.2. Cross-platform Solutions for Mobile UI Test Automation - Clustering

The difference between them all lays in the progression described below:

• OEM automation tools are the most robust one between the API-based tools. They come with a sufficient set of functions to build the commonly used test patterns, but in case of Apple UIAutomation the scripting is too wordy. They also are limited to the one platform.

• Wrappers are cross-platform solutions. Appium tool is the only wrapper so far. The vendors of the several other tools have adopted their cloud testing labs (with real devices) to run Appium tests. Wrappers add some additional weak points per platform, per script language, per environment. It means that if something does not work then the

¹³ http://appium.io/

¹⁴ <u>https://developer.xamarin.com/guides/testcloud/introduction-to-test-cloud/</u>

¹⁵ http://www.tricentis.com/tricentis-tosca-testsuite/tosca-mobile-plus/

¹⁶ http://docs.telerik.com/teststudio/test-studio-mobile/overview

¹⁷ http://www.keynote.com/solutions/testing/mobile-testing

¹⁸ http://www.ranorex.com/test-automation-tools.html

¹⁹ https://docs.experitest.com/display/public/SA/Manually+Instrumenting +iOS+Applications
²⁰ <u>http://www.sikuli.org/testing.html</u>
²¹ <u>http://docs.testplant.com/ePF/using/epf-getting-started-ios-gateway.htm</u>

issue could be related exactly with the code that does wrapping, while the same command would work in the OEM automation tool.

Name	Device Support	Cloud Support	Scripting Languages	Native/ Hybrid	Web	Costs
Appium ¹³	X	<u>-+/-</u>	Java, Ruby,	X	X	Free/
Appium	Λ	(Simulator	Python, PHP,	Λ	(comes	
		(Sinulator only)	JavaScript, C#		with	pay for
		onry)			wrapper)	cloud
Xamarin	Х	X (can run	C#, Ruby	X	+/-	Paid
Test	21	Appium)	Cir, Ruby	21	(need to	1 and
Cloud ¹⁴		rippium			wrap the	
olouu					website	
					into native	
					app)	
Tosca	Х	Х	Through IDE,	Х	X	Paid
Mobile+ ¹⁵		(private	VB, C#,		(comes	
		cloud with	VBScript		with	
		deviceConn			wrapper)	
		ect123 by				
		MobileLabs				
)				
Telerik	Х	Х	Through IDE,	+/-		Paid
Test Studio			C#	(hybrid		
Mobile ¹⁶				are not		
				supported)		
Device	Х	X (can run	Through IDE,	Х		Paid
Anywhere ¹⁷		Appium)	Java			
Ranorex ¹⁸	Х	Х	Through IDE,	Х		Paid
a m .19			C#, VB.Net			
SeeTest ¹⁹	Х	X (can run	Through IDE,	Х		Paid
		Appium)	C#, Java, Perl,			
~			Python, Ruby			
Sikuli ²⁰	iOS -	-	Java, Python,	Х	Х	Free
	simulator		Ruby,			
E D (²¹	only	37	JavaScript	37	N/	D 11
EggPlant ²¹	Х	Х	SenseTalk,	Х	Х	Paid
			Java, C#,			
			Ruby			

 Table 5.3. Cross-platform Solutions for Mobile UI Test Automation – Characteristics

The solutions that need the 3rd party library integration into the source code have the same pros and cons as wrappers do. But there are two additional weak points:

• The code of the app under tests is changed in comparison to the release version. It increases the probability of app working differently when it is built for the automated testing purposes. Of course, the same applies for all automation solutions, because they all interfere into the app under test in some way. But there is more trust that this interference is properly handled when the OEM solution is used.

• It is not possible to access the system modal windows/ popovers and device functions from these libraries. Test framework can access them only by calling the methods of OEM automation API.

Image based tools can simplify the recognition of UI object in a short term, however, having them as the only solution has the following cons:

• Early automation is hardly possible – with agile software development approach it is very possible that image slices are not yet available, while functionality is already there.

• UI can vary not only per platform, but also per device type (phone vs. tablet). In case of image based tools it will increase the test creation and maintenance costs.

• Adjusting/ refreshing UI up to new OS guidelines most probably will trigger more test maintenance effort than it would be needed for API-based solutions.

To conclude, tests written using the tools that are using image pattern recognition of the UI object are quite fragile in comparison to API-based solution, while having the image comparison for assertion in some cases is the only way to go for UI level tests. More detailed overview of the solutions can be discovered in Kulešovs (2017).

6. tTap Extension for Apple UIAutomation

6.1. Apple UIAutomation Capabilities and Limitations

Before choosing UIAutomation as a target test automation tool more deep analysis of its capabilities and limitations was performed. The identified functional blocks were divided into several levels: application level, OS level, device level, device/ OS level combined, and framework level. (Kulesovs et al., 2015)

6.1.1. Application Level

On application level UIAutomation is capable to interact with all native UI elements, as well as interact with custom developed UI elements that either extend or customize the native UI elements or are totally custom designed UI elements that extend the top UI elements – UIView or UIViewController.

The tool also supports all native gestures, however the native pinch to zoom gesture does not work on simulator starting already from iOS 7. The support of custom gestures is quite limited. Custom gestures can be simulated only if they can be performed with a single drag between two points. It also is not possible to simulate the complex drag gesture between more than two points. So, if it is needed to simulate the drawing of a curve, this could be achieved only by performing a large set of drag gestures, finishing each of them as if the finger was taken out from the device screen. Even if result will look mostly the same (the circle is drawn), this is achieved with completely different internal logic.

It is possible to change the app settings (stored in setting property list file) during the test run. However, if it is needed to change the setting before the test run (e.g. the setting that is applied when the application starts), then the setting should be adjusting during the build process or app to be restarted after setting is changed during the first run. Restarting the app is possible only by stopping the current test run and by starting another one.

There is an option to simulate memory warnings when app runs on the simulator. However this function is not accessed in UIAutomation out of the box.

UIAutomation understands UIWebView structure, so it is possible to interact with the web content. However, web apps to be built in into the native app to test them using UIAutomation. The tool is not supposed to run the default Safari browser (and any other built-in app), that is why there is a very basic custom web browser app with a single web view is created by the community to test the web apps on iOS. The issue is that if web app has complex JavaScript inside then web view should also implement much more functionality than a single default web view.

6.1.2. OS Level

On OS level the tool is able to send the app to background for a definite amount of time. However, it is not possible to switch between the apps, even if they both are custom built. Switching between apps could be useful, for example, if there is an intention to check how Open In works for the app under test. It is not possible to manipulate with push notifications as well.

6.1.3. Device Level

On device level it is possible to perform the orientation change of the device. This is very important feature for mobile UI tests. There is also an ability to simulate the pressing of device buttons to change volume, lock and unlock the device (without the passcode), simulate as if it is being shaken.

6.1.4. Device/ OS Level

On the combined device/ OS level the tool allows to manipulate with the location services, i.e. to set the latitude, longitude, altitude, course, moving speed, etc. However, the tool is missing the support of switching on/ off the WiFi connection, as well as the ability to change the connectivity speed. The tool is also missing the ability to manipulate with the restrictions, privacy settings, and region formats. Simulating the interruptions like receiving the phone call or SMS is also not possible.

Often there is also a need to add and/ or remove the images from Photo app and contacts from Contacts app to create different preconditions for test execution or to perform the cleanup before or after the test. It is not possible to perform such actions out of the box. However, there is a possibility to execute the tasks on the host Mac machine that from which tests are executed.

6.1.5. Framework Level

On framework level UIAutomation is missing unit test style notations test runner. It also is not able to search for the element within the whole element tree. It searches only within the first level children out of the box. There is a set of functions to check some basic conditions like if element is present, however there are no out of the box wait statements, as well as more complex conditions are needed to check if the tool can interact with the element.

```
Kuļešovs et al.
```

The tool is capable to take a screenshots while the test being executed. But there is no built in comparison inside it. Typing on the keyboard also fails from time to time when characters from the different keyboards are being typed in (e.g. letters and numbers and/ or special symbols, letters in capital and in narrative).

6.1.6. Summary

To summarize, Apple UIAutomation can perform the most of the basic functions that can be executed on the iOS device. However, there are still several limitations in terms of functionality and framework usability. The described capabilities and limitations of the tool are aggregated in Table 6.1 and Table 6.2.

Table 6.1. The Capabilities of Out of the Box Apple UIAutomation Capabilities

Application
Interact with all built-in UI elements
Interact with custom developed UI elements
Support for all built-in gestures
Web-views
Changing app settings
OS
Sending app to background/ foreground
Device
Simulating device buttons pressing (i.e. volume, etc.)
Orientation change
OS/ Device
Manipulation with location services

Table 6.2. The Limitations of Out of the Box Apple UIAutomation Limitations

Application
Support for custom developed gestures
Support for complex dragging gesture (more than two points)
Low-memory warnings
OS
Switching between apps
Open app from push notification
Device
-
OS/ Device
Switching on/ off WiFi connection
Changing the connectivity speed
Restrictions and Privacy Settings
Region Formats
Interruptions
Add/ remove images from Photos app
Add/ remove contacts from Contacts app
Framework
Unit test style notation

78

Limited assertions capabilities
Searching within the whole UI elements tree
Image comparison
Wait conditions
Robustness of keyboard typing
Limited logging/ debugging capabilities

6.2. Choosing the Right Tool for the Environmental Context

To choose the tool to automate UI tests with, we have done the following:

- Investigated each solution from Table 5.1 and Table 5.2.
- Took into account the weak points set of each solution described in section 5.
- Took into account the particular environmental options within our company.
- The environmental context can be described as:

• There is no need for cross-platform support in our case, because the majority of the apps we produce are iOS native apps (while we already are creating them using the cross-platform Xamarin²² tool taking into account the possible future requests). It is so, because this is what enterprise clients currently need, as shown by the statistics.

• We want to limit the investigation time of searching which of the components has failed if something does not work.

• We want to decrease the probability of something does not work after the consecutive update of the tool and/ or native automator.

The image-comparison based tools are quite powerful solutions, but due to the very agile nature of mobile apps development, at least in our company, when UI and UX can change dramatically in a couple of weeks we have excluded this option due to the probable maintenance effort.

The arguments above led to choosing native Apple UIAutomation as a target solution to automate UI tests. When choosing the tool we have acknowledged the limited debugging capabilities of UIAutomation due to the own, non standard JavaScript environment where tests are executed. When we were considering the options, Apple XCTest was not available yet.

6.3. The Rise of tTap

6.3.1. Introduction

When doing the first proofs of concepts in UIAutomation we took a look at both of the extensions mentioned in section 5.2.1.1. We decided to take Tuneup JS as a core extension, because CSS-style of mechanic.js did not seem convenient for us with Java background. We have also made a study of what is missing in the original UIAutomation framework (see section 6.1). During the extensive test automation process it appeared that we need the different sets of commands in comparison to Tuneup JS to make the test automation process more convenient. That is why we started to cut, rewrite, and extend

²² <u>http://xamarin.com/</u>

Tuneup JS extension that resulted into new extension creation that we call $tTap^{23}$ – target tap.

The main reason for this title is that almost all actions within the extension are executed in absolute coordinates of the device while still operating on the UIAElements (UIView and UIViewController) level. The device (or simulator) is called *target* in UIAutomation context. The decision to work in absolute coordinates was made to overcome several issues that we will describe in a course of this section. It is worth mentioning that tTap extension is distributed under MIT license²⁴.

6.3.2. Solution Details

The goal of tTap is to overcome the limitations of the original UIAutomation aggregated in Table 6.2. The details on overcoming each limitation are described below. Overcoming of limitations on the application level, OS level, and device level is achieved through triggering the execution of AppleScripts to manipulate the built-in OSX application that can interact with an iOS device on the host machine (the machine to which iOS device is connected during the test execution). On framework level the limitations are overcome through triggering the execution of shell scripts to perform, for example, image comparison, etc. New JavaScript functions are written to extend and enhance the existing framework as well.

6.3.3. Application Level

The end to end support for custom developed gestures could not be achieved at all. The only option is to integrate the custom library inside the application under test that will call the same method execution as if custom gesture is performed. However, this is a big overhead, because normally there should be the way to use the same functionality using the simple tap, long press, or swipe gesture. The support for complex dragging gesture (more than two points) can also be achieved through the custom library integration into the application under test. But, again, there should be very strong argument for doing so.

It is possible to simulate low-memory warnings when app runs on simulator. This could be achieved through calling the respective function from the simulator menu (see Fig. 6.1.). To do this in automatic way tTap contains function that runs AppleScript that "clicks" this menu item using the hotkeys.

6.3.4. OS Level

It is not possible to switch between the apps from UIAutomation framework. The issue is that even if another app is opened by calling it through URL schema, UIAutomation does not see it. It can be attached only to one application during the test run.

Quite often apps perform some internal navigation when they are opened from push notification. In general, it is possible to manipulate with alerts from UIAutomation. However to make such test repeatable is very hard, because the notification is sent through Apple Push Notification Network in real time. That is why the time when the notification arrives to the device is unpredictable.

80

²³ <u>https://github.com/ivans-kulesovs/tTap</u>

²⁴ http://opensource.org/licenses/MIT

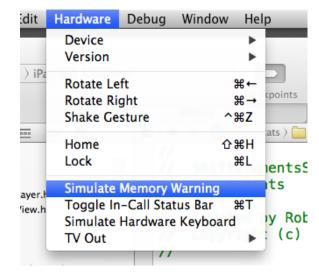


Fig. 6.1. Triggering low-memory warning on the simulator.²⁵

6.3.5. Device/ OS Level

There is no way to switch on/ off WiFi connection or any other connection through UIAutomation. However, it is possible to share the WiFi connection from Mac machine and use it on iOS device where tests are executed. In such case, the shared WiFi could be switched on/ off by using one of the tools from Apple Developer toolset called LinkConditioner (see Fig. 6.2). There is also an option to simulate the connection speed of different connection types like EDGE, 3G, etc., as well as to simulate the bad network conditions or to create custom network conditions (see Fig. 6.3).

AppleScript to manipulate the LinkConditioner tool from UIAutomation is a part of tTap framework. The AppleScript contains the functions to run/ close LinkConditioner, to select the definite network condition profile, and to switch the selected profile on/ off.

There are several ways how to deal with the initial set of photos and/ or contacts for the automatic test presetup. One way is to use tTap commands that run specific AppleScripts to manipulate with photos and contacts. It is possible to open such OSX apps as Photos and Contacts and use their functionality to add/ remove data from the device. This presetup is especially important in case of there are several apps being tested on the same device. Another way is to build and install the custom app that adds/ removes such kind of data from the device.

In general, it is possible to simulate the different types of interruptions like phone call, receiving of SMS, etc. However, this is not needed, because it is up to operating system, not up to the app how to deal with such kind of interruptions. Receiving of the phone call for the app is the same as switching it to background, nothing more.

It is impossible to manipulate with restrictions and privacy settings within the app. The permissions to the photos and contacts can be asked by the app only once.

²⁵ <u>http://stackoverflow.com/questions/5323515/unable-to-simulate-invoking-applicationdidreceivememorywarning</u>

Afterwards OS prevents app from asking them again, even if user uses the functionality of the app where these permissions are needed. User can change the restrictions only manually in app or OS settings after giving the response on the first prompt. These security features of OS make it impossible to automatically check the app behavior when there are some restrictions set for the app, because it limits the testing of positive cases afterwards without a manual intervention.

To have a constant automatic check of app behavior under different regional and time setting is possible only if tests run on the multiple devices with different regional and time settings being preset.

Show All	Network Link Conditioner	Q
Network Link Conditioner	Profile: Wifi DNS Delay: None Downlink Bandwidth: 40 mbps Packets Dropped: 0% Delay: 1 ms	t Uplink Bandwidth: 33 mbps Packets Dropped: 0% Delay: 1 ms
Click the lock to make	changes.	Manage Profiles

Fig. 6.2. Switching on/ off WiFi connection via LinkConditioner.

* Preset Promies	(
3G, Average Case 3G, Good Connectivity	Downlink Bandwidth :	222		Kbps ‡
3G, Lossy Network	Downlink Packets Dropped :	10	%	
Cable Modem DSL	Downlink Delay :	225		ms
Edge, Average Case Edge, Good Connectivity	Uplink Bandwidth :	100	1 Destination	Kbps ‡
Edge, Lossy Network	Uplink Packets Dropped :	10	%	
Wifi, Average Case Wifi, Good Connectivity Wifi, Lossy	Uplink Delay :	225		ms
Custom Profiles	DNS Delay :	150		ms
+ - o *				

Fig. 6.3. Creating new network conditions profile.

82

6.3.6. Framework Level

The modified test runner from Tuneup JS is used to run the tests. Tests follow the unit tests style convention. The test suite is wrapped into JavaScript function. There is a separate file where these "test suite" functions are called in the definite order. We have extended the test runner with the possibility to ignore the definite tests and make some tests dependant from another tests result (e.g. not to run the test if the precondition is not achieved). The advanced assertions capabilities are taken from Tuneup JS without modification.

UIAutomation allows searching for the element only within one node of the UI elements tree. tTap implements the recursive search by accessibility identifier from the root node or from the definite parent. The idea is taken from Penn (2013).

As already mentioned, almost all actions are made on device (target) level. This is the closest way how touches occur in reality. Gestures are executed on the target using the calculated center point of UIAElement in absolute coordinates. iOS recognizes the object at these coordinates and go through the responder chain searching the element that executes the actions responding to the definite gesture, as shown in Fig. 5.4. This solves the following:

• There are cases when UIAutomation does gesture on the wrong coordinates if the command is called exactly from the UIAElement. It more often occurs with the system windows like email controller or some context menu, especially if app is created using some cross-platform solutions like Xamarin. We have not searched for the reason, but this workaround works perfectly.

• By default UIAutomation does tap at (0, 0) point of UIAElement, while the real user tends to tap to the center of the object in the most cases.

• This led to the idea of creation such convenient and often used function as *UIAElement1.tDragAndDrop(UIAElement2)* where the object on top of which to drop the current object is set as a parameter.

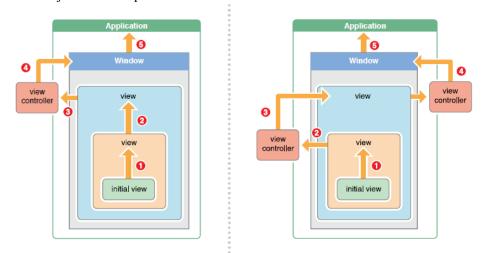


Fig. 6.4. The examples of responder chain in iOS. (Apple Developer, 2015)

UIAutomation has quite limited logging capabilities that, taking into account the JavaScript object nature of UI elements, is not sufficient for proper debugging. We refer to debugging here, because there is no other way to debug than doing extensive logging in UIAutomation environment. There is more extensive logging mechanism available in tTap extension.

tTap framework comes with a set of different wait conditions like waiting for the ability to tap the element, waiting till it is visible, waiting till it reaches the specific position on the screen, etc.

We have adjusted the default inter key delay of the keyboard from 0.03 seconds till 0.2 seconds that makes typing more robust, because it constantly failed when switching between the keyboard types (e.g. numeric, capital letters), especially when running tests on CI machine, and even more often when run on simulator.

The authors have improved the image comparison solution that comes with Tuneup JS. It used to fail when there were some tens of files on the desktop, because the screenshots temporary were stored there. We also have rewritten it to perform the comparison of images with an option to set the similarity threshold. Now its robustness does not rely on the number of files on the desktop. It is worth mentioning that UIAutomation itself allows only capturing the screenshot.

6.3.7. Summary

All identified UIAutomation limitations were thoroughly examined during the rise of tTap extension. Majority of the limitations where workaround is possible are solved in tTap, for some of the limitation there is a clear way how to deal with, however the solution is not reliable enough (i.e. can have the issue with tests repeatability) due to the objective infrastructure limitations (e.g. opening app from Push Notification), or such tests are not needed at all because they can be compensated with other tests (e.g. testing of interruptions). Some of the limitations can be solved by setting up the advanced infrastructure (e.g. running tests on multiple devices with different regional and date/ time settings). There are also limitations left not solved, because of OS or UIAutomation restriction by purpose (e.g. changing the restrictions, privacy settings, etc.). Various of the framework limitations like unit test style notations, advanced assertions, logging, and debugging capabilities are solved after ability of creating UI tests has been added to Xcode 7 itself in XCTest framework. The status of overcoming Apple UIAutomation limitations is shown in Table 6.3.

To summarize, in comparison to OEM Apple UIAutomation, with tTap extension during automated tests execution it is possible:

• To switch on/ off the connectivity to the WiFi shared from the automated tests execution host that allows checking how app works in offline, as well as to simulate the network interruption during the online activity.

• To perform the image-based comparison of the whole screen or its part with an etalon. Sometimes, it is the only way to perform the assertion. In other cases it could be less complex to make such kind of assertion than the logical one.

• To add images and contacts from/ to Photos and Contacts apps. This allows making the repeatable test data of such type before test execution and to clean up test data in test tear down block.

Limitation	Status in XCTest	Status in tTap
Application		
Support for custom developed	×	×
gestures		
Support for complex dragging	×	×
gesture (more than two points)		
Low-memory warnings	×	✓ (on
		simulator only)
OS		-
Switching between apps	×	×
Open app from push notification	×	×
Device		
-		
OS/ Device		-
Switching on/ off WiFi connection	×	✓
Changing the connectivity speed	×	✓
Restrictions and Privacy Settings	×	×
Region Formats	×	× (can run
		tests on multiple
		devices)
Interruptions	×	Not needed
Add/ remove images from Photos	×	\checkmark
app		
Add/ remove contacts from	×	\checkmark
Contacts app		
Framework		-
Unit test style notation	✓	✓
Limited assertions capabilities	✓	✓
Searching within the whole UI	×	\checkmark
elements tree		
Image comparison	×	\checkmark
Wait conditions	×	\checkmark
Robustness of keyboard typing	×	\checkmark
Limited logging/ debugging	✓	\checkmark
capabilities		

Table 6.3. The Status of Overcoming Apple UIAutomation Limitations

Other than that several improvements to the framework that simplifies it or makes it more robust have been developed. They are:

- unit test style notations;
- advanced assertion capabilities;
- searching within the whole UI elements tree;
- various wait conditions;
- improved keyboard typing robustness;
- advanced logging/ debugging capabilities.

In order to perform a stress testing tTap also allows simulating low-memory warning, however only on simulator, not on the real device.

7. Ideal Cross-Platform Mobile UI Test Automation Tool Proposal

The analysis of the mobile UI test automation tools from the fifth section, the analysis of the possibilities and limitations of the out of the box Apple UIAutomation, and creating the solutions for these limitations united in tTap frameworks resulted into the proposal of the ideal cross-platform mobile UI test automation tool creation. The device control is achieved with EggPlant image-based tool instrument –custom Springboard. More matured Appium tool (that being a wrapper on top of native automators has the best concept for cross-platform support) is used for test automation itself. Its part for iOS apps automation is extended with the solutions united in tTap framework. Android UIAutomator and Microsoft Coded UI Tests capabilities and limitations still to be investigated and solved if possible. Even Appium wraps only iOS and Android native automators, there already a project called Winium²⁶ started that wraps Coded UI Tests as well. The proposal is schematically depicted in Fig. 7.1.

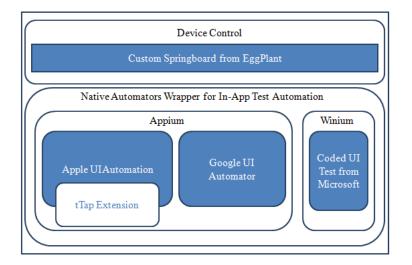


Fig. 7.1. The ideal cross-platform mobile UI test automation tool architecture.

Using custom Springboard from EggPlant to control the device is cleaner and most efficient way in case of iOS. Jail-break is a no go for device control at all. Using wrapping concept is the cleanest way possible for cross-platform UI test automation solution.

²⁶ <u>https://github.com/2gis/Winium.StoreApps.CodedUi</u>

8. Conclusions and Discussions

The study consists of six main parts. Each part adds value to the software testing field in general and to the field of the mobile software testing in particular.

The first part concentrates on the inventory and structuring of testing ideas and terms. It has resulted into discovering of eight classes of the testing ideas. Initiation of such process has helped to understand the need of making the clear definition of such terms as testing approach, testing method, and testing techniques that has been achieved using the solution made by Anthony in the field of language teaching. Testing methods and techniques have been united under black box, white box, and in-operational testing approaches. Structuring of the ideas have also made it possible to schematize and visualize the software testing on meta-level, defining the relation between such concepts as testing strategy, testing tactics, testing schools, testing mission, testing wision, different (organizational and project-wide) contexts, testing approach, testing method, testing technique, testing plan, etc. Uniting various software testing processes under software testing of the software testing ideas and terms has been done for the first time as well.

In the second part the aspects of mobile applications functional testing are investigated. iOS was chosen as a target platform for investigation, because it is the current market leader in the enterprise world. The literature review of both academic and multivocal literature was performed. The majority of the sources selected for the review, both academic and multivocal, were published during the last seven years period.

The results of SLR are mostly related to general mobile applications testing aspects like limited resource utilization, orientations, localizations, etc., while the results of MLR provided the needed details of iOS application testing aspects (like definite restrictions and privacy settings, iOS accessibility features, etc.), as well as identified some new aspects like IAP, date/ time settings, etc. The identified aspects were divided between 4 large clusters: *Environment, Application Lifecycle, Inside the Application*, and (functional or performance aspects of) *UI/ UX*.

The third part looks into the functional security testing of iOS applications. While application security in most cases is tested by the security specialists, it is possible and is much cheaper to verify that security mechanisms provided by OS vendor are used as much as possible (if the nature of the app needs it, of course) before giving the app to them. This often is neglected in favor of time to market rush. These mechanisms are usage of the secure network protocols, data base encryption, and locking the application data. Another functional security testing part is to check and eliminate the leftovers of development and testing activities in the productive build of the app. The examples of leftovers are settings files and code that reads them or performs the action based on the setting value. This issue is not discussed in the literature at all. The authors got the details through the own studies while breaking the apps.

The solutions for mobile UI test automation are discovered and categorized in the fourth part. These solutions can be divided into three parts: OEM tools, API-based tools, and image comparison based tools. API-based tools can also be divided into two groups: wrappers (tools that wrap the native automators) and tools that need 3rd party library integration into the application code. All non OEM tools are cross-platform tools.

Tests written using the tools that are using image pattern recognition of the UI object are quite fragile in comparison to API-based solution, while having the image comparison for assertion in some cases is the only way to go for UI level tests.

The study of the (mobile) automation tools available on the market has been performed by practitioners many times, for sure. However, most of them are not publically available, but those who are do not provide any thorough categorization of the tools.

The rigorous study of the capabilities and limitation of Apple UI Automation has been done for the first time. This is reflected in the sixth section. These capabilities are divided among several levels: application, OS, device, and OS/ device. UIAutomation capabilities are: interact will all built-in UI elements; interact with custom developed UI elements; support for all built-in gestures; web-views; changing app settings; sending app to background/ foreground; simulating device buttons pressing (i.e. volume, etc.); orientation change; manipulation with location services. Out of the box UIAutomation limitations are: support for custom developed gestures; support for complex dragging gesture (more than two points); low-memory warnings, switching between apps; open app from push notification; switching on/ off WiFi connection; changing the connectivity speed; restrictions and privacy settings; region formats; interruptions; add/ remove images from Photos app; add/ remove contacts from Contacts app; unit test style notation; limited assertions capabilities; searching within the whole UI elements tree; image comparison; wait conditions; robustness of keyboard typing; limited logging/ debugging capabilities.

All limitations were analyzed and solutions were provided for those that do not require to jailbreak device or perform any other hacking of the OS or device. These solutions are united in tTap framework – the extension for Apple UIAutomation. The following limitations are solved in tTap framework: low-memory warnings (on simulator only); switching on/ off WiFi connection; changing the connectivity speed; add/ remove images from Photos app; add/ remove contacts from Contacts app; unit test style notation; limited assertions capabilities; searching within the whole UI elements tree; image comparison; wait conditions; robustness of keyboard typing; limited logging/ debugging capabilities.

The investigations, analysis, and tTap solution creation from the fifth and sixth sections led to the ideal cross-platform mobile UI test automation tool proposal. The device control is achieved with EggPlant image-based tool instrument – custom Springboard. Appium tool (together with Winium spin off for Windows) as a wrapper concept is used for test automation itself. Its part for iOS apps automation is extended with the solutions united in tTap framework.

This is also the fact, that the mobile applications field is not mature enough yet. It is even less mature in terms of testing. The authors investigations on the iOS apps testing aspects, iOS apps functional security aspects, and mobile UI test automation, including the creation of tTap framework and making the proposal of the ideal cross-platform mobile UI test automation tool, focuses on this issue and makes the field a bit more mature.

Acknowledgments

The research was partly developed under the University of Latvia contract no. AAP2016/B032 "Innovative information technologies".

References

- Addey, D. (2013). iOS Devices, available at http://daveaddey.com /postfiles/AgantReleaseChecklist2013.pdf
- Amalfitano, D., Fasolino, A., Tramontana, P., Amatucci, N. (2013). Considering context events in event-based testing of mobile applications.Proc. IEEE 6th Int. Conf. Softw. Testing, Verification and Validation Workshops, Luxembourg.
- App Quality Alliance (2013). Testing Criteria for iOS Apps, available at http://www.appqualityalliance.org/files/AQuA_testing_criteria_ for_iOS_for_v1.0%20final%2022_oct_2013.pdf
- Apple Developer (2012). UI Automation JavaScript Reference for iOS, available athttps://developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAuto mationRef/
- Apple Developer (2015). Event Handling Guide for iOS, available at https://www.hitpages.com/doc/6281807121088512/1
- Arnicans, G., Romans, D., Straujums, U. (2013). Semi-automatic Generation of a Software Testing Lightweight Ontology from a Glossary Based on the ONTO6 Methodology. Frontiers in Artificial Intelligence and Applications. Databases and Information Systems VII: Selected Papers from the Tenth International Baltic Conference, vol. 249, IOS Press, pp. 263-276.
- Arnicans, G., Straujums, U. (2012). Transformation of the Software Testing Glossary into a browsable Concept Map. International Conference on Engineering Education, Instructional Technology, Assessment, and E-learning (EIAE 12); International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 12).
- Barzdin, J., Bicevskis, J., Kalninsh, A. (1974). Construction of Complete Sample Systems for Program Testing, Ucenye Zapiski Latv. Gos. Univ., vol. 210, pp. 152-188.
- Barzdin, J., Bicevskis, J.,Kalninsh, A. (1977). Automatic Construction of Complete Sample System for Program Testing, in IFIP Congress.
- Beizer, B. (1990). Software Testing Techniques, 2nd ed., New York: Van Nostrand Reinhold Co.
- Beizer, B. (1995). Black-Box Testing: Techniques for Functional Testing of Software and Systems, New York: John Wiley & Sons, Inc.
- Bicevska, Z., Bicevskis, J.,Oditis, I. (2015). Smart technologies for improved software maintenance.Federated Conference on Computer Science and Information Systems (FedCSIS), Lodz, Poland.
- Bicevskis, J., Bicevska, Z., Oditis, I. (2016). Self-management of information systems. 12th International Baltic Conference on Databases and Information Systems, Riga.
- Citrix (2015). Citrix Data Reveal New Global Trends in Consumer and Enterprise Mobility, available at http://www.citrix.com/news/announcements/feb-2015/citrix-data-reveal-new-global-trends-in-consumer-andenterprise-.html
- Dantas, V., Marinho, F., da Costa, A., Andrade, R. (2009). Testing requirements for mobile applications. Proc. 24th Int. Symp. Comput. and Inform. Sci., Guzelyurt.
- Diebelis, E., Bicevskis, J. (2013). Software Self-Testing. Frontiers in Artificial Intelligence and Applications, vol. 249: Databases and Information Systems VII, pp. 249-262.
- Eston, T. (2012). Android vs. Apple iOS Security Showdown, available at http://pittsburgh.issa.org/Archives/Android-vs-iOS-MayUpdate.pdf
- Franke, D., Elsemann, C., Kowalewski, S., Weise, C. (2011). Reverse engineering of mobile application lifecycles. Proc. 18th Work. Conf. Reverse Eng., Limerick.
- Franke, D., Kowalewski, S., Weise, C. (2012). A mobile software quality model.Proc. 12th Int. Conf. Quality Softw., Xi'an, Shaanxi.
- Franke, D., Kowalewski, S., Weise, C., Prakobkosol, N. (2012). Testing conformance of lifecycledependent properties of mobile applications. Proc. 5th Int. Conf. Softw. Testing, Verification and Validation, Montreal.

Gao, J., Xiaoying, B., Wei-Tek, T., Uehara, T. (2014). Mobile application testing: a tutorial. IEEE Computer, vol. 47, no. 2, pp. 46-55.

Good Technology (2013). Good Technology™ Mobility Index Report Q4 2013, available athttps://media.good.com/documents/rpt-mobility-index-q413.pdf

Goodenough, J. and Gerhart, S. (1975). Toward a Theory of Test Data Selection, IEEE Transactions on Software Engineering, vol. 1 (2), pp. 156-173.

- Gorbans, I., Kulesovs, I., Buls, J., Straujums, U. (2015). The Myths about and Solutions for an Android OS Controlled and Secure Environment. Environment. Technology. Resources. Proceedings of the 10th International Scientific and Practical Conference., Rezekne.
- Hall, G. (2011). Exploring English Language Teaching: Language in Action, New York: Routledge.

Haller, K. (2013). Mobile testing. ACM SIGSOFT Softw. Eng. Notes, vol. 38, no. 6, pp. 1-8.

- ISO (2011). ISO/IEC 25010:2011.
- ISTQB (2012). Standard glossary of terms used in Software Testing.
- Kalninsh, A. and Borzovs, J. (1981). Inventarizacija idej testirovanija programm, Riga, Latvia: Latv. Gos. Univ.
- Kaner, C., Basch, J., Pettichord, B. (2001). Lessons Learned in Software Testing: A Context-Driven Approach, New York: John Wiley & Sons, Inc.
- Kaner, C., Falck, J., Nguyen, H. (1999). Testing Computer Software, 2nd ed., Wiley & Sons, Inc.
- Khalid, H. (2013). On identifying user complaints of iOS apps. Proc. 35th Int. Conf. Softw. Eng., San Francisco, CA.
- Kim, H.-K. (2012). Mobile applications software testing methodology. Commun. In Comput. and Inform. Sci., vol. 342, pp. 158-166.
- Kitchenham, B., Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering, EBSE Tech. Rep.
- Kulesovs, I. (2015). iOS Applications Testing.Environment. Technology. Resources. Proceedings of the 10th International Scientific and Practical Conference., Rezekne, Latvia.
- Kulešovs, I. (2017), Mobile Applications Testing, PhD thesis, University of Latvia, Riga, Latvia.
- Kulešovs, I., Arnicane, V., Arnicans, G., Borzovs, J. (2013). Inventory of Testing Ideas and Structuring of Testing Terms. Baltic J. Modern Computing, vol. 1, no. 3-4, pp. 210-227.
- Kulesovs, I., Susters, A., Keiduns, K., Skutelis, J. (2015). Automated Testing of iOS Apps: tTap Extension for Apple UIAutomation.3rd International Conference on Horizons for Information Architecture, Security and Cloud Intelligent Technology: Programming, Software Quality, Online Communities, Cyber Behaviour and Business (HIASCIT), Sanremo, Italy.
- Kumaravadivelu, B. (2006). Understanding Language Teaching: From Method to Postmethod, Mahwah, NJ: Lawrence Erlbaum Associates, Inc.
- Land, R. (2012). iOS Accessibility A Useful Guide For Testing, available at http://www.rosiesherry.com/2012/09/02/ios-accessibility-auseful-guide-for-testing/
- Marinho, E. H.,Resende, R. (2012). Quality factors in development best practices for mobile applications. Proc. Computational Sci. and Its App., Salvador de Bahia, Brazil.
- Markets and Markets (2014). Heterogeneous Mobile Processing & Computing Market by Component (processor, GPU, DSP, connectivity), Technology Node (45NM-5NM), Application (Consumer, Tele-communication, Automotive, MDA, Medical), & Geography -Forecast & Analysis to 2014 – 2020, available at http:// www.marketsandmarkets.com/Market-Reports/heterogeneous-mobileprocessing-computing-market-173926586.html
- Muccini, H., Di Antonio, F., Esposito, P. (2012). Software testing of mobile applications: challenges and future research directions. Proc. IEEE 7th Int. Workshop Automation of Softw. Test, Zurich.
- Myers, G. (1979/2004). The Art of Software Testing, 2nd ed., Hoboken, NJ: Wiley & Sons, Inc.

- Nearsoft (2013). Testing iOS Apps for Tough Network Conditions, available at http://nearsoft.com/blog/testing-ios-apps-for-tough-networkconditions/
- Neglected Potential (2013).iOS Testing mind map 1.2 Now with more stuff, available at http://www.neglectedpotential.com/2013/10/ios-testing-mindmap-1-2/
- Oditis, I.,Bicevskis, J. (2015). Asynchronous Runtime Verification of Business Processes: Proof of Concept. International Journal of Simulation - Systems, Science & Technology, vol. 16, no. 6, pp. 1-11.
- Oditis, I.,Bicevskis, J. (2015). Asynchronous Runtime Verification of Business Processes. Proceedings - 7th International Conference on Computational Intelligence, Communication Systems and Networks, Riga.
- Ogawa, R. and Malen, B. (1991). Towards rigor in reviews of multivocal literatures: applying the exploratory case study method. Review of Educ. Research, vol. 61, no. 3, p. 265–286.

OpenSignal (2015). Android Fragmentation Visualized, available at http://opensignal.com/reports/2015/08/android-fragmentation/

- Penn, J. (2013). Test iOS Apps with UI Automation: Bug Hunting Made Easy. Dallas, Texas: The Pragmatic Bookshelf.
- Pettichord, B. (2007). Schools of Software Testing, available at http://www.prismnet.com/~wazmo/papers/four_schools.pdfPound, P. (2013). Tips For Accessibility Testing Of iOS Apps, available at
- Pound, P. (2013). Tips For Accessibility Testing Of iOS Apps, available at http://patstapestry.wordpress.com/2013/05/24/tips-foraccessibility-testing-of-ios-apps/
- Pressman, R. (2005). Software Engineering: A Practitioner's Approach, 6th ed., Singapore: McGraw-Hill.
- Rauhvarger, K.,Bicevskis, J. (2008). Towards a Semantic Execution Environment Testing. Scientific Papers, University of Latvia, vol. 733, pp. 38-52.
- Research and Markets (2014). Mobile Cloud Market by Application(Gaming, Entertainment, Utilities, Education, Productivity, Business & Finance, Social Networking, Healthcare, Travel & Navigation), & By User (Enterprise User, Consumer)-Worldwide Market Forecast and Analysis (2014 - 2019), available at http://www.researchandmarkets.com/research/7pj4cv/mobile cloud
- Scott, A. (2012).Introducing the software testing ice-cream cone (anti-pattern), available at http://watirmelon.com/2012/01/31/introducing-the-softwaretesting-ice-cream-cone/
- SmartBear (2014). Testing iOS Applications, available at http://blog.smartbear.com/mobile/testing-ios-applications/

Sommerville, J. (2007). Software Engineering, 8th ed., Harlow, Essex: Pearson Education Ltd.

- TestElf (2013). We Find These Common Bugs When Testing iOS Apps, available at http://blog.testelf.com/post/56341438836/we-find-these-commonbugs-when-testing-ios-apps
- Tom, E., Aurum, A., Vidgen, R. (2013). An exploration of technical debt. Journal of Systems and Software, vol. 86, no. 6, pp. 1498-1516.
- uTest (2013).The Essential Guide to iPhone & iPad App Testing, available at http://qawiki.devsmm.com/wp-content/uploads/2014/10/ uTest_Whitepaper_The_Essential_Guide_to_iOS_App_Testing.pdf

Received February 11, 2018, revised March 28, 2018, accepted March 29, 2018