UNIVERSITY OF LATVIA
FACULTY OF COMPUTING

IVANS KUĻEŠOVS

MOBILE APPLICATIONS TESTING

Doctoral Thesis

Area: Computer Science and Information Technologies
Sub-area: Software Engineering

Scientific Advisor: Dr. habil. dat., Prof.
Juris Borzovs

RIGA 2017

ABSTRACT

The mobile conquers the world. The need in a comprehensive and systemized multi-edge testing approach is rising along with mobile apps becoming even more complex, while there is still some chaos present in the general testing terminology and clear classification of terms. The thesis gives an overview of software testing on the meta-level, providing the theoretical background related to the classification of terms on testing techniques, methods, and approaches.

As a leader in enterprise market, Apple iOS has been chosen as a target mobile platform for the study. Aspects that influence functional testing of iOS apps in particular, and mobile – in general, were investigated by the author. The thesis also exposes the security capabilities and development/ testing leftovers that often are neglected and not cleaned up in favor of time to market rush.

A separate chapter of the thesis deals with mobile UI test automation tools investigation and clustering. The capabilities and limitations of Apple UIAutomation are discovered in the last chapter. Solutions aimed at overcoming the limitations of out of the box UIAutomation are united in tTap framework developed by the author. The practical usage experience gathered during tTap framework creation is shared in the thesis as well. The practical usage experience includes the difference between executing tests on a real device and on a simulator, examples of situations when image comparison is the only or the most efficient assertion possible, examples of test scenarios where change in connectivity is needed, and more.

The analysis performed on mobile UI test automation tools united with the solutions used in tTap framework resulted in the ideal cross-platform mobile UI test automation tool proposal.

The whole thesis has a practical flavor. All mobile testing related information has been verified or gathered during the real mobile software development projects execution.

The thesis consists of five chapters, 102 pages, 30 figures, and 9 tables.

Keywords: mobile applications testing, iOS, test automation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

Acronym or Abbreviation	Meaning
UI	User Interface.
BYOD	Bring Your Own Device - policy of permitting employees to bring personally owned mobile devices.
(KV) Charts	Karnaugh-Veitch Charts – black-box testing technique.
SLR	Systematic Literature Review - a type of literature review that collects and critically analyzes multiple research studies or papers.
MLR	Multivocal Literature Review – a type of literature review that collects and critically analyzes accessible, but non-academic writings on the topic.
Springer	Springer Links – digital library.
IEEE	IEEE Xplore – digital library.
ACM	ACM Digital Library
OS	Operating System.
IAP	In-App Purchase – technology that allows buying items inside the app.
ROM	Read-only Memory - a file containing the executable instructions (a system image) of an Android OS and affiliated apps.
UWP	Universal Windows Platform - a platform-homogeneous application architecture created by Microsoft.
CI	Continuous Integration - the practice of merging all developer working copies to a shared mainline several times a day.

INTRODUCTION

In 1975, the first theoretic foundation of testing by Goodenough & Gerhart [1] was published. A year before that the first publication on software testing was published in Latvia by Barzdin et al. [2], but an enriched version of it was presented in 1977 [3]. Since those times, theory and practice of testing have evolved quite significantly through emergence of testing activists (Myers [4], Beizer [5] [6], Kaner [7] [8], Bach [8], Pettichord [8] [9], Black [10], 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 [11], the mobile cloud market is expected to be worth US $46,90 billion by 2019, while the research by Markets and Markets [12] 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 [13], iOS holds 64%, and according to Good Technology [14], iOS holds even 73% market share of all enterprise mobile devices. According to the same study by Good Technology [14], 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 [15]. 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 [16], 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 are 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 [17].

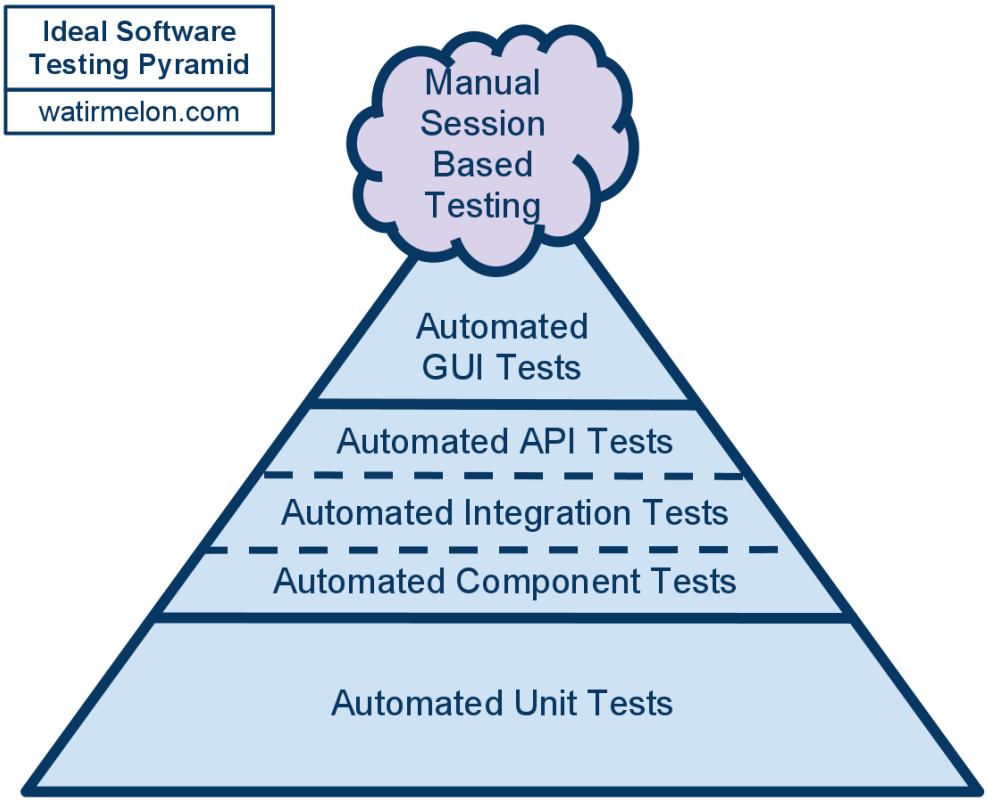


Fig. 1. Automated test coverage model per test level. [17]

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 author and his colleagues.

The whole thesis has a practical flavor. All mobile testing related information has been verified or gathered during the real mobile software development projects execution.

Thesis consists of five chapters. In the first chapter the systemized overview of software testing is given. The proposal to organize the relation among software testing terms approach, method, and technique is part of the chapter as well. In the second chapter the systematic review on mobile applications testing aspects through the prism of iOS mobile operating system is presented. In the third chapter a highlight on mobile functional security testing is provided. The fourth chapter describes the study on existing mobile UI test automation solutions and their pros and cons. The capabilities and limitations of Apple UIAutomation tool are described in the fifth chapter. The details on tTap extension for Apple UIAutomation that overcomes part of the OEM automator limitations are recorded in the same chapter. The chapter concludes with the proposal of ideal cross-platform mobile UI test automation tool based on the information gathered and solutions prepared during the thesis rise.

THE AIM AND TASKS OF PHD THESIS

The tasks of our PhD thesis are as follows:

1) To give a systemized overview of the software testing field.
2) To gather and systemize aspects (i.e. features and/ or limitations) that influence testing of functional suitability, performance efficiency, compatibility, reliability, maintainability, and portability of iOS native business applications to fill the gap that exists in the current academic literature.
3) To perform a comprehensive analysis of the gathered aspects to optimize their appliance in real iOS applications testing strategies.
4) To point out security capabilities of iOS that are often neglected in favor of time to market rush.
5) To provide insight into the solutions available on the market for mobile apps UI test automation, in particular for iOS apps, and help to choose the right one depending on the environmental context of a testing organization.
6) To analyze limitations of a native iOS UI test automator and create a solution that provides workarounds for them where applicable.
7) To come up with a suggestion of an ideal cross-platform mobile UI test automation tool.

THESES OF PHD THESIS

1) The testing terms approach, method, and technique have a vague meaning in the existing testing literature.

2) Knowing a defined and detailed set of aspects that influence testing of iOS native business applications will increase the quality of such apps through increasing the test coverage.
3) Various iOS security capabilities are neglected in favor of time to market rush. Such situation should be eliminated.
4) Apple UIAutomation has various limitations. The existing workarounds help to overcome one group of limitations, however it is impossible to overcome others on non-jailbroken device or without integrating a custom library into the app source code.
5) An ideal cross-platform mobile UI test automation tool can be created by means of uniting concepts that already exist in the market.

RESEARCH METHODS

1) Systematic reviews of academic and multivocal literature sources have been performed in order to get a comprehensive overview of the software testing field and to gather and systemize aspects that influence testing of iOS native business applications.
2) Documentation analysis, static analysis, and dynamic analysis data collection techniques have been applied to gather and process the data needed for the comprehensive analysis of:
   - aspects that influence testing of iOS native business applications;
   - the neglected security capabilities of iOS;
   - mobile UI test automation tools;
   - features and limitations of Apple UIAutomation.
3) Various visual modeling techniques have been used to report the data analysis results of a systemized overview of the software testing field, of the aspects that influence testing of iOS native business applications, and of the features and limitations of Apple UIAutomation.
4) Various experiments have been performed to expose the neglected security capabilities that allow breaking the app via a reverse engineering method.
5) A number of experiments have been performed to test the workarounds found to overcome the limitations of Apple UIAutomation.

SCIENTIFIC NOVELTY

1) The main result of the thesis is tTap framework development that overcomes the limitations of native Apple UIAutomation automator for iOS mobile operating system.

2) Besides that the variety of methodical improvements were made in the software testing theory field:
   - Our own systemized overview of software testing is provided.
   - A new systematization of software testing terms approach, method, and technique is given.
   - A detailed list of aspects that influence testing of iOS native business applications, which has not been published before, was created and systemized.
3) The proposal of an ideal cross-platform mobile UI test automation tool has been provided.

PRACTICAL VALUE

1) Our own systemized overview of software testing, as well as a new systematization of software testing terms approach, method, and technique can be applied while teaching software testing professionals and computer science students. This should give a clearer and more comprehensive understanding of the software testing field for them.
2) A complete, systemized, and detailed list of aspects that influence testing of iOS native business applications can be used for creating more thorough testing strategies. These strategies should be applied for testing the real iOS apps.
3) Exposing the neglected security capabilities of iOS should help software development organizations to create more secure apps.
4) The study on solutions for mobile applications UI test automation should help test professionals to choose the right one, depending on the environmental context of a testing organization.
5) tTap extension for Apple UIAutomation is already used for UI test automation of iOS apps in C.T.Co [1] software development company. Testing specialists from other organizations could use it as well if an appropriate marketing campaign is performed.

APPROBATION OF PHD THESIS

The research results are described and published in the following papers or book chapters:

1) Ivans Kulesovs et al. (2015). The Multi-Edge Approach for iOS Applications Testing. In Cipolla Ficarra, F. et al. (Ed.), New Perspectives from User Interfaces and Semantic Web: Information Quality, Advanced Interdisciplinary Applications and Combination

[1] http://www.ctco.lv/

of the Technologies Challenges, Blue Herons, Bergamo, Italy, pp.77 – 107 (accepted for publication). Involvement – 80% (text writer, main driver, discussions participant).

2) Ivans Kulesovs, Aigars Susters, Kirils Keiduns, Janis Skutelis. *Automated Testing of iOS Apps: tTap Extension for Apple UIAutomation*. In proceedings of "3rd International Conference on Horizons for Information Architecture, Security and Cloud Intelligent Technology: Programming, Software Quality, Online Communities, Cyber Behaviour and Business (HIASCIT)", July 9 – 10, 2015, Sanremo, Italy, pp. 4 – 22. Involvement – 80% (text writer, main driver, discussions participant).
3) Ivans Kulesovs (2015). *iOS Applications Testing*. In "Environment. Technology. Resources. Proceedings of the 10th International Scientific and Practical Conference." vol. 3, Rezekne, Latvia, pp. 138 – 150. [Scopus]
4) Ivans Kuļešovs, Vineta Arnicane, Guntis Arnicans, Juris Borzovs (2013). *Inventory of Testing Ideas and Structuring of Testing Terms*. In "Baltic J. Modern Computing", vol. 1, No. 3 -4, pp. 210 – 227. Involvement – 50% (text writer, discussions participant).
5) Imants Gorbans, Ivans Kulesovs, Uldis Straujums, Jānis Buls. *The Myths about and Solutions for an Android OS Controlled and Secure Environment*. In "Environment. Technology. Resources. Proceedings of the 10th International Scientific and Practical Conference." vol. 3, Rezekne, Latvia, pp. 54 – 64. Involvement – 30% (text reviewer, discussions participant). [Scopus]
6) Jānis Buls, Imants Gorbans, Ivans Kulesovs, Uldis Straujums (2016). *The Adaptation of Shamir's Approach for Increasing the Security of a Mobile Environment*. In "Baltic J. Modern Computing", vol. 4, No. 1, pp. 51-58. Involvement – 15% (text reviewer, discussions participant). [Web of Science]

The research results were presented at the following conferences:

1) Ivans Kulesovs, Aigars Susters, Kirils Keiduns, Janis Skutelis. *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), July 9 – 10, 2015, Sanremo, Italy.
2) Ivans Kulesovs. *iOS Applications Testing*. 10th International Scientific Practical Conference "Environment. Technology. Resources." 18-20 June 2015, Rezekne, Latvia.
3) Juris Borzovs, Ivans Kuļešovs, Vineta Arnicāne, Guntis Arnicāns. *An Attempt to Systemize Software Testing Concepts*. 5th International Workshop "Data Analysis Methods for Software Systems", December 5 -7, 2013, Druskininkai, Lithuania.

4) Imants Gorbans, Ivans Kulesovs, Uldis Straujums, Jānis Buls. *The Myths about and Solutions for an Android OS Controlled and Secure Environment*. 10th International Scientific Practical Conference “Environment. Technology. Resources.” 18-20 June 2015, Rezekne, Latvia.

1. INVENTORY OF TESTING IDEAS AND STRUCTURING OF TESTING TERMS

1.1. Inventory of Testing Ideas

The author and his colleagues have performed the inventory of testing ideas [18]. It was inspired by [19]. 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 [20]. As a result, a map showing the linkage between the testing terms and their relation to the definite 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 [21] [22]. The same color scheme, as shown for the ideas classes, is used to distinguish the terms parent in the resulting graphs.

1.2. 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 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. 1.1.

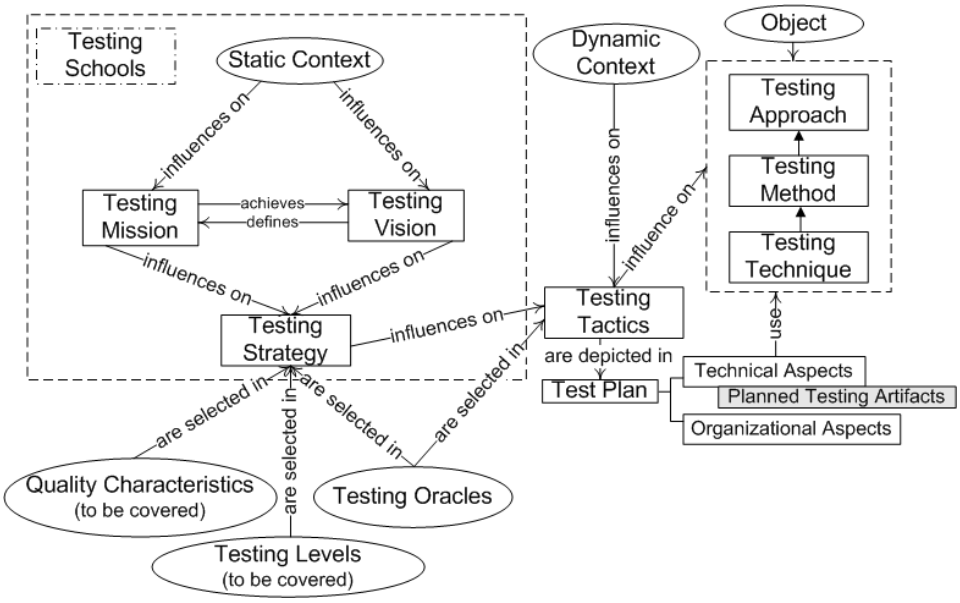


Fig. 1.1. Software testing review on meta-level.

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. Testing oracles are in details discussed in the section 1.5.1.

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.

1.3. Software Testing Dichotomies

There are many dichotomies exist in software testing. Some of them clearly belong to definite testing school. Others are opposable because of other reasons, for example project phase. It is worth mentioning that dichotomies mentioned below, despite their difference, make good testing when used together proportionally.

The dichotomy we should start with is ***testing vs. debugging***. The goal of testing is to discover the defect while the goal of debugging is to find why the defect occurs. Some schools see debugging as a job of software developer only, but nowadays it is more common for good test engineer to investigate the root cause of the defect by himself or together with a software developer.

The most known testing dichotomy is ***black-box testing vs. white-box testing***. The difference between them is the point of view on the knowledge of the internal structure of the software that test engineer takes when designing the test cases.

Functional testing vs. non-functional testing is another important testing dichotomy. Functional testing "*verifies a program by checking it against ... design document(s) or [functional] specification(s)*" [7]. Non-functional testing checks software against its non-

functional requirements where non-functional quality characteristics are addressed. System testing is different from functional testing because it "*validate[s] a program by checking it against the published user or system requirements*" [7].

Another quite old dichotomy is ***manual testing vs. automated testing***. Return on investment is taken into consideration when testing is automated, as it requires skilful workforce and additional scripting and maintenance effort. Still, only part of the testing can be automated. UI automation is often used for regression testing, while unit and integration tests can be written in advance to development.

These two testing ideas are very different by their nature: ***scripted testing vs. exploratory testing***. Scripted testing can show the thoroughness of the testing to stakeholders, while exploratory testing can find failures that hardly could be discovered when using scripted testing, because it is sometimes even hard to imagine the appropriate test cases before investigating the behavior of the new functionality under test.

Another dichotomy consists of two of the oldest testing ideas: ***verification vs. validation***. Controversy ***contract vs. client happiness*** is closely connected to the testing missions represented above, thus, depending on this, different testing strategies are chosen. Verification evaluates if product meets the requirements that usually are part of the contract while validation check if product satisfy the clients (or other stakeholders) expectations, i.e. makes them happy.

Positive testing vs. negative testing dichotomy both parts are necessary if there is an aim to make testing as complete as possible. Positive testing tends to prove that software behaves in the way it is supposed to. Negative testing shows that software does not do that it is not supposed to.

Testing of design vs. testing of implementation identifies different testing needs depending on the software project phase. Thus, different testing tactics can be used during each phase. Testing of design also uncovers the idea that testing should be started as early as possible.

Static testing vs. dynamic testing dichotomy intersects with previously mentioned dichotomy. Testing of designs is always a static testing, i.e. testing process without executing the software itself. Testing of implementation (except the review of the code) in most cases is a dynamic testing, i.e. testing of the running software.

Hierarchical vs. big bang are different approaches of the integration testing. There are two hierarchical integration testing approaches: bottom-up and top-down. When bottom-up approach is used then testing is started from the components on the lowest level and goes up to the testing of the integration of the next level components. Integration testing between top level components is the first point of the top-down approach. It goes to the lower level components testing afterwards till the lowest level is reached. On the contrary, integration on all levels occurs simultaneously when big bang integration testing approach is used.

The last, but not least software testing dichotomy that we would like to mention is ***traditional testing vs. agile testing***. Agile school has completely different mission then other ones. Agile testing proves that system under test works as expected. It discovers the role of software engineer in test as the great automation specialist and the main participant of the test driven development.

1.4. Testing Schools

Testing society distinguishes five testing schools [9]. 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. Usage of schools can vary within the organization from project to project, but it is often hard to move the whole organization from one school to another.

The *analytic school* assumes that software is a logical artifact. It concentrates on technical aspects, and it is keen on the white-box testing. Analytic school is associated with academia institutions, and it is assumed to be the most suitable for safety-critical and telecom software.

The *standard school* assumes that testing should be very well planned in advance and managed. According to this school, testing main goal is to validate that software meets contractual requirements and/or governmental standards using the most cost-effective model, thus it is mostly applied for governmental and enterprise IT products. Requirements traceability matrix is the most common testing artifact for the school. Software testing can be seen like assembly line through V-model prism. IEEE standards' boards and testing certifications are the most valued institutions by this school.

The *quality school* prefers "Quality Assurance" over "Testing". Thus testing defines and controls the development processes. QA manager or test lead is like a gatekeeper who can decide if software is ready or not. ISO and CMMI are the most valued institutions for followers of this school.

The *context-driven school* concentrates about (skilled) people and their collaboration. The goal of the context-driven testing is to find bugs that can bother any of the stakeholders. What to

test right now is defined according to the current situation in the project. Test plans to be constantly adapted based on the test results. Exploratory testing is an exemplar technique of this school. Context-driven testing is mostly applied for the commercial and market-driven software. The Los Altos Workshop on Software Testing held by Cem Kaner and Brian Lawrence are thought to be the main events of this school.

The *agile school* main postulate is that tests must be automated. Testing answers the question if user story is done (and system under test works as expected). Test-driven development is one of the agile testing school paradigms, thus automated acceptance tests are demonstrative exemplar of the school.

It is worth mentioning that categorization of beliefs and testing goals into testing schools helps testers to understand and to evaluate each other experience through the prism of the specific organizational context.

1.5. Testing Strategy

There are many things to be covered in the testing strategy that define the overall approach for testing. Here we only look into the details of its most important aspects that are noticeable on the software testing meta-level.

1.5.1. Testing Oracles

Testing oracles can be divided into three major groups based on their purpose. Groups of oracles and representatives per each group are shown in Fig. 1.2.

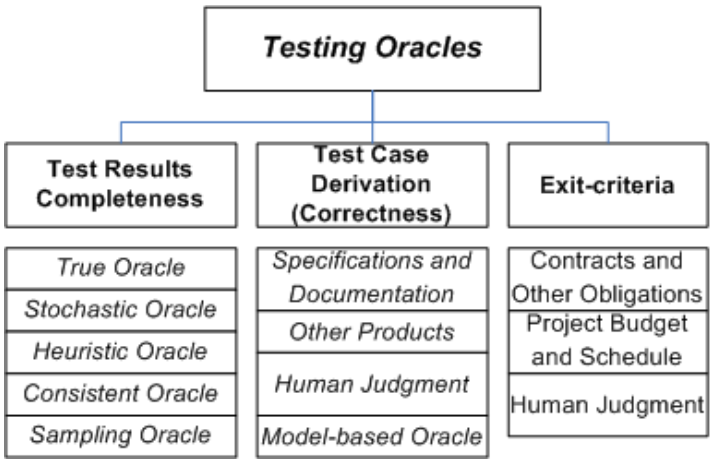


Fig. 1.2. Testing Oracles. [23]

Test results completeness oracles are differentiated based on the completeness of the set of the expected test results. There are five main types of test result completeness oracles. They are [23]:

- True oracles – they have the complete set of expected test results.
- Stochastic oracles – they verify a randomly selected sample.
- Heuristic oracles– they can verify the correctness of some values and the consistency of other values.
- Consistent oracles – they verify current test run results with previous test run results (regression).
- Sampling oracles – they select the specific collection of inputs or results.

They all have their advantages and disadvantages, as well as their cost decreases in a top-down manner, but speed increases in the same manner.

Test case derivation (correctness) oracles are differentiated based on the source test cases and expected results are derived from.

Exit-criteria oracles define when testing can be finished. The most common, but not complete testing exit-criteria are:

- All planned test cases are executed (Contracts and other obligations).
- All high and critical priority bugs are fixed (Contracts and other obligations).
- All planned requirements are met (Contracts and other obligations).
- Scheduled time to finish testing has come (Project budget and schedule oracle).
- Test manger has signed off the release (Human judgment oracle).

It is worth mentioning that multiple oracles of each group are often used together depending on the software project phase.

1.5.2. Quality Characteristics

There are 8 quality characteristics shown in the new revision of ISO/IEC 9126 standard - ISO/IEC 25010 [24]. All quality characteristics are depicted in Fig. 1.3 and Fig. 1.4.

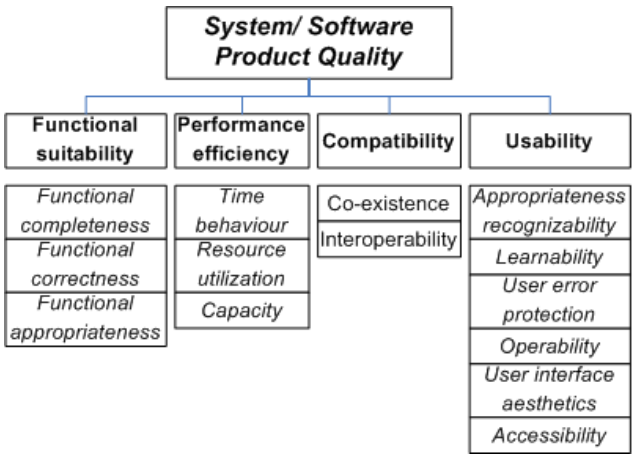


Fig. 1.3. Product Quality Model. [24]

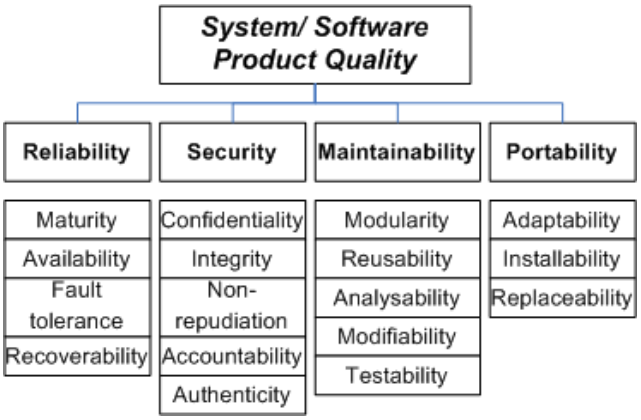


Fig. 1.4. Product Quality Model (continued). [24]

Functional testing is a testing of functional suitability characteristic. Almost all formal testing methods and techniques are concentrated around functional suitability quality characteristic. They are especially related to the functional correctness and functional completeness sub-characteristics.

1.5.3. Testing Levels

There are four main *testing levels* differentiated in the software development project. Their applicability differs based on the project phase and the scale of the object under test. These levels are [10]:

- Unit Testing – testing of single component on the code level; it is usually performed by developers.
- Integration Testing – testing of cooperation of several components; comparison with expected result can be done both on the code level and manually by human; can be performed either by developer, or by tester.

- System Testing – testing of the whole complete system; usually is performed by tester.
- Acceptance Testing - testing of the whole system to verify that it meets some contract obligation and/ or satisfies users' expectation about the software product; usually is performed by the customer.

All levels starting from integration testing can be scaled out till the system of systems testing when product consists of or is dependent on multiple systems.

1.6. Testing Tactics

Testing tactics can differ depending on the phase of the project and other changeable circumstances of the environment. *Testing tactics* should be consistent with the testing strategy. Thus tactics often are chosen within the static boundaries of the influencer schools. Appropriate testing approaches, methods, and techniques should be selected for micro testing goals fulfillment and should be depicted in the test plan. *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. We have structured testing methods and techniques under *black-box* and *white-box* approaches. The borders of grey-box testing approach are quite ambiguous, and methods and techniques under this approach are not formally described yet in the testing theory. They do not have settled definitions in the testing practice as well.

It is worth mentioning that we respect other software testing systematization concepts, for instance, division of all methods and techniques under 4 coverage approaches (Graph Coverage, Logic Coverage, Input Space Partitioning, and Syntax-Based Testing) by Ammann & Offutt [25], but we still have not found the explicit difference between testing approach, method, and technique in other concepts.

1.6.1. Testing Artifacts

Software testing usually produces testing artifacts mentioned below:

- Test Data – multiple sets of values to be used as inputs for testing definite functionality often combined into one file.
- Test Script – code that substitutes user activity and/or interaction with software UI.
- Test Case – consists of preconditions, steps, inputs, and expected results to test some part of the functionality.
- Test Scenario – test case with higher level of abstraction that depicts scenarios in which user is considered to use the software.

- Test Suite – the set of test cases or test scenarios for given functionality or testing type (i.e. regression, smoke, or sanity).
- Test Plan – document that depicts testing tactics to test definite software product in the definite testing run; often consists of the test suites to be executed and the testing approach to be used.

Traceability matrix is the example of cross-referring document that can be used to depict the relations between test cases/test scenarios/ test suites (depending on the scale) and requirements.

Test harness is a virtual, to testing related artifact that consists of many aspects to make testing under given conditions and configurations possible. It can consist of the specific IT infrastructure, tools, big samples of test data, etc.

Despite the fact that testing artifacts mentioned above to be produced during the whole testing process lifecycle, the high level description of the approach to be used to produce them to be defined in the testing strategy.

1.6.2. 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 [6] 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**) …" [6]; "**[T]est execution technique:** The **method** used to perform the actual test execution, either manual or automated" [20]. Other authors, such as Kaner et al. [7] [8], Pressman [26], and Sommerville [27] 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 & Gerhart was published. In 1963 Anthony provided "much needed coherence to the conception and representation of elements that constitute language teaching:" (as cited in [28])

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 & Rogers (as cited in [28]) 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. [29]

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. 1.5.

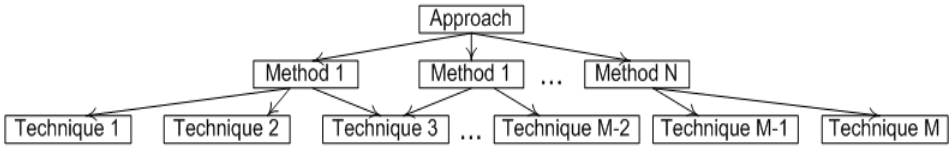


Fig. 1.5. Relation between approach, method, and technique.

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”*.

1.6.3. 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. 1.6.

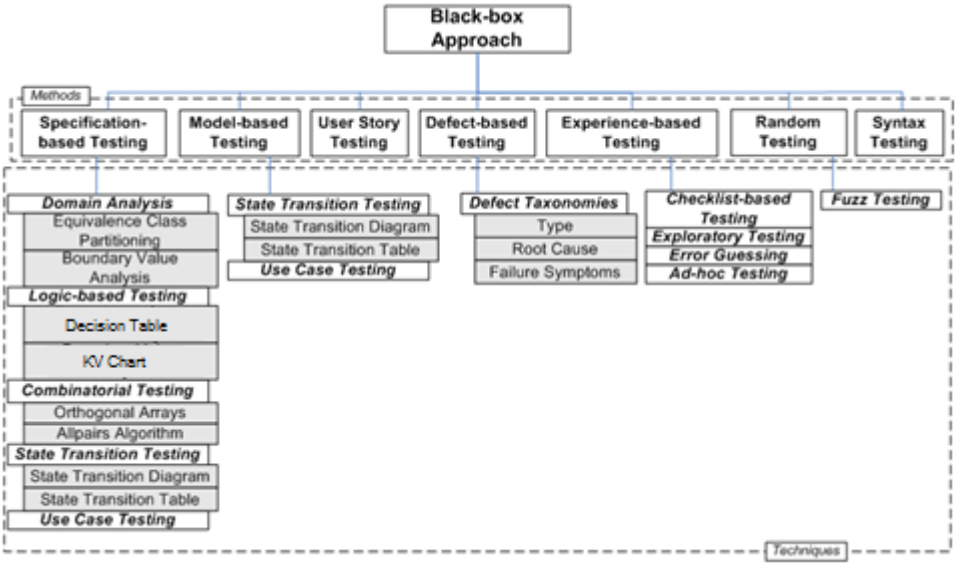


Fig. 1.6. Black-box Approach.

Specification-based testing is a testing method which includes all formal test case design techniques. As can be derived from the name of the method, specification (or requirements) documents are used as a source for test case design. Formal test case design techniques or groups of techniques are Domain Analysis, Logic-based Testing, Combinatorial Testing, State Transition Testing, and Use Case Testing.

Domain analysis group consists of two closely connected testing techniques: Equivalence Class Partitioning and Boundary Value Analysis. The first technique defines the group (class) of inputs that produces the same output. The second technique checks the boundary values of the equivalence classes.

Logic-based testing group consists of two testing techniques: Decision Tables and Karnaugh-Veitch (KV) Charts. They all are used when combination of different inputs results into specific output. They are used for checking business logic and user interface. According to Copeland [30], a decision table consists of conditions, combinations of every condition alternatives that result into single rules, actions, and actions occurrence under every rule. It is

worth mentioning that cause-effect graphing can also be used for designing decision tables according to Myers [4].

KV charts are used to simplify the Boolean algebra expressions. They were introduced by E. Veitch in 1952 and improved by Karnaugh in 1953. They allow decreasing the amount of calculation needed through humans' pattern-recognition capability [5]. From the author experience usage of decision tables is more common technique from these two in the field of business application testing, especially nowadays.

State Transition Testing is a group of techniques that are used when some part of the functionality of the system can be represented as “finite-state machine”. Finite state machine is an abstract machine that has finite states, that can be in only one state at once, and whose transitions from one state to another are triggered by some event or condition. There are two common techniques that are used for state transition testing: State Transition Diagrams and State Transition Tables. State transition diagram is a schematic representation of machine’s states and transitions between them. State transition table is more complete and systematic way of representation of the same machine’s states and transitions. Only valid state-transition combinations are depicted in the state transition diagrams, while all possible state-transition combinations are covered in the state transition tables that can be required for testing the safety-critical software. [10]

Combinatorial Testing is a group of testing techniques that are most often used for testing combinations of configurations or input parameters. The most popular techniques are Orthogonal Arrays and Allpairs Algorithm. Orthogonal array is a two-dimensional array that has an interesting property – “all the pairwise combinations will occur in all the column pairs” [30]. This part of discrete math was introduced into testing field by Tatsumi in 1987 [31]. Allpairs algorithm invented by Bach allows achieving the coverage of testing of all pairs combination with less steps when input parameters have different number of possible values [8].

Use Case Testing is a technique that allows to test system’s functionality that is described as a use case. Use case is a type of quite detailed specification that concentrates on user (or another system) interaction with the system under test to complete some specific task or to deliver some other business value. It often has a main, the most commonly used flow and extensions or some special cases. The test scenario for main flow and every extension or special case should be created when use case testing is performed. Use case can be described using natural language or depicted using different modeling languages, for example, UML.

Model-based testing is a testing method which unites testing techniques that use similar software, or software prototype, or software usage models as the basis for test cases design. The

main representatives of this method are previously described state transition testing and use case testing (when use case is described using some of the modeling language).

User story testing method includes acceptance testing techniques in combination with exploratory testing techniques that are described later. User story is a way of a non-detailed software specification that describes it using the mask "As an <actor> I want (or need) <action> so that <achievement>" (in practice, sometimes <achievement> part is not formally specified). User story must come to the development team together with acceptance criteria to align the constraints of the business value to be delivered. User stories are mostly used when software is developed using such Agile software development practices as Scrum, Kanban, and XP. Acceptance tests are executed to verify if implemented user story meets the acceptance criteria. More thorough testing using exploratory testing techniques is performed after acceptance criteria is met. Sometimes, depending on the complexity of the system, usage of more formal testing techniques also takes place.

Experience-based testing method unites less formal testing techniques, but some of them are still very powerful when are used by professionals. These techniques are Checklist-based Testing, Exploratory Testing, Error Guessing, and Ad-hoc Testing.

Very high level checklist of quality attributes or items that are important for the system under test is used for checklist-based testing. Such list should be constantly improved to cover things that are important to some of stakeholders or are parts of some regulation standard (for example, operating system UI guidelines) while product is evolving during the development process.

Test engineer intuition and experience to evaluate the test results are the basis of the exploratory testing technique. The design of new test cases occurs on the fly using the information discovered from the testing of the software itself. Exploratory testing to be productive must be performed in definite time frames and the scope of testing must be defined in advance. Test charters are often used to make these two "musts" possible and also show the productivity of the testing session to the stakeholders by notifying its results. Such exploratory testing management was developed by Jonathan and James Bach in 2000. They named it session-based testing, but we suppose that exploratory testing without clearly defined objectives and time frames is ad-hoc testing that is the least formal testing technique of the experience-based testing method. Usage of ad-hoc testing technique should be avoided. [10]

Error-guessing is a testing technique that uses most common programming errors as test case basis. Examples of such errors are null pointers, division by zero, wrong types of parameters, etc. Even if tester does not have knowledge of programming she will often discover such errors while testing the software and will reuse this experience afterwards. That is why this

technique is part of experience-based testing method. In most cases error-guessing is used as informal supplementary of formally scripted testing techniques.

Defect-based testing method uses the knowledge about defects taxonomies for test cases design or selection. According to Beizer [5], there are eight categories to be used for defects classification: Functional, System, Process, Data, Code, Documentation, Standards, and Other. There are also five supplementary categories to be used for defects housekeeping: Duplicate, Not a problem, Bad Unit, Root cause needed, Unknown. Black [10] uses the same defects classification. From the author experience this method can be hardly used independently for test cases design. It can only point out which test cases may lead to more defects discovery based on the historical data if it is available. What is more, such analysis of defects is quite expensive and such bookkeeping options are supported by only few tools by default.

Random testing method uses randomly generated inputs from the definite subset as test data. It can be a powerful method for functional testing when operational profile (input domains) of the system and effective oracle are available. In such cases systems are tested with condition that whole test fails if it fails on at least one of the inputs. But in real situation the options mentioned above are hardly available. Even if uniform distribution can be applied to the input values, it is very hard to substitute the effective oracle for outputs. That is why random testing is mostly used for reliability testing of the complex systems. It can prove that system can work without failures for given amount of time [32]. When reliability of the system is tested with totally random input values it means that Fuzz Testing technique is applied.

Syntax testing is a static, black box testing method for testing syntactic specification of a system's (or protocol's) input values. "Anti-parser" can be used to compile the grammar to produce "structured garbage". This "structured garbage", that can contain misplaced or missing elements, illegal delimiters, and so on, is used to test how object under test behaves when inputs deviate from the defined syntax. [5]

1.6.4. 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. 1.7.

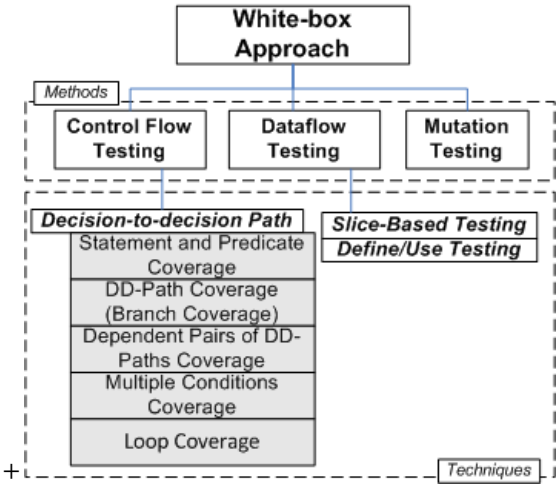


Fig. 1.7. White-box Approach.

Control flow testing concentrates about testing the sequence of the statements in which system under test operates. There are two main programming paradigms that influence the statements' sequence execution. They are conditions and loops. The main technique of control flow testing is called Decision-to-Decision Path Testing [33]. Decision-to-Decision path testing technique uses program graph to represent all possible statements (graph nodes) and conditions (graph edges). Coverage of different code aspects can be checked when using this technique.

Dataflow testing method concentrates about the points of program graph where variables receive values and where these variables are used. Thus dependent pairs of the DD-paths coverage of previously mentioned Decision-to-Decision Path Testing technique is most efficient exit criteria for such testing method while the whole lifecycle of the variable is monitored.

Mutation testing method is used to prove that the set of unit tests that pass actually is correct and complete. Mutation (i.e. wrong peace of code) is introduced into the program itself. For example, operators or commands execution order can be changed, or even some code can be removed. If unit tests still pass after mutation introduction then it means that some of the unit tests are wrong or that mutated code is never executed.

Some static testing techniques are used for software code testing. They differ based on the formality and thoroughness of the process. Code review is often used to improve the overall quality of the code and to educate less experienced developers. This process helps to deliver more qualitative and tested code from development to testing right at the moment, but educative aspects help to improve the quality of the code for the future deliveries. Inspections and walkthroughs are used when there is less time available to conduct the static testing process.

1.6.5. 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, self-testing, runtime verification of business process execution 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. [34] [35] The relation between the *in-operational* testing methods and techniques is shown in Fig. 1.8.

Execution environment testing checks that software surrounding environment matches "the requirements regarding its execution, for instance, OS version, configuration file and registry entries, regional settings, etc." [36] This automated check can be done both during the deployment and during the run-time.

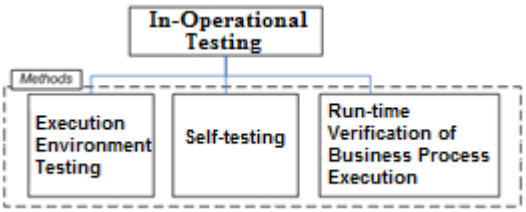


Fig. 1.8. In-Operational Testing.

Self-testing ensures "that the software is working correctly at any point of its life cycle". [37] This check can be automated that gives the method the pro-active flavor.

(Asynchronous) runtime verification of business process execution is a method that gives an opportunity to verify that business processes run as expected without interrupting the business processes already being executed inside the live system. [38] [39]

2. MOBILE APPLICATIONS TESTING ASPECTS

2.1. Apple iOS

2.1.1. Introduction

According to the different studies [11] – [14] iOS devices hold the major market share among the corporate workers.

With the growth of platform abilities applications become more complex [15] 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 chapter and study in general.

Despite the fact that the topic being hot, there are only some academic studies [40], [41], [42] performed that systemize the generic aspects that should be taken into consideration when testing the mobile applications without specifying the platform. Other studies – [43] and [44] 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 author 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 [45]) sources in the field of iOS testing [46].

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 [24]. 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. The goal of the chapter is to eliminate the gap that currently exists between academic and non-academic sources in the field of iOS applications testing, as well as to provide the sufficient details for practitioners to make their iOS applications testing strategy more complete and solid.

2.1.2. Research Methodology

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 [47]. Fig. 2.1 shows the stages of sources selection for the whole review process applied in this paper.

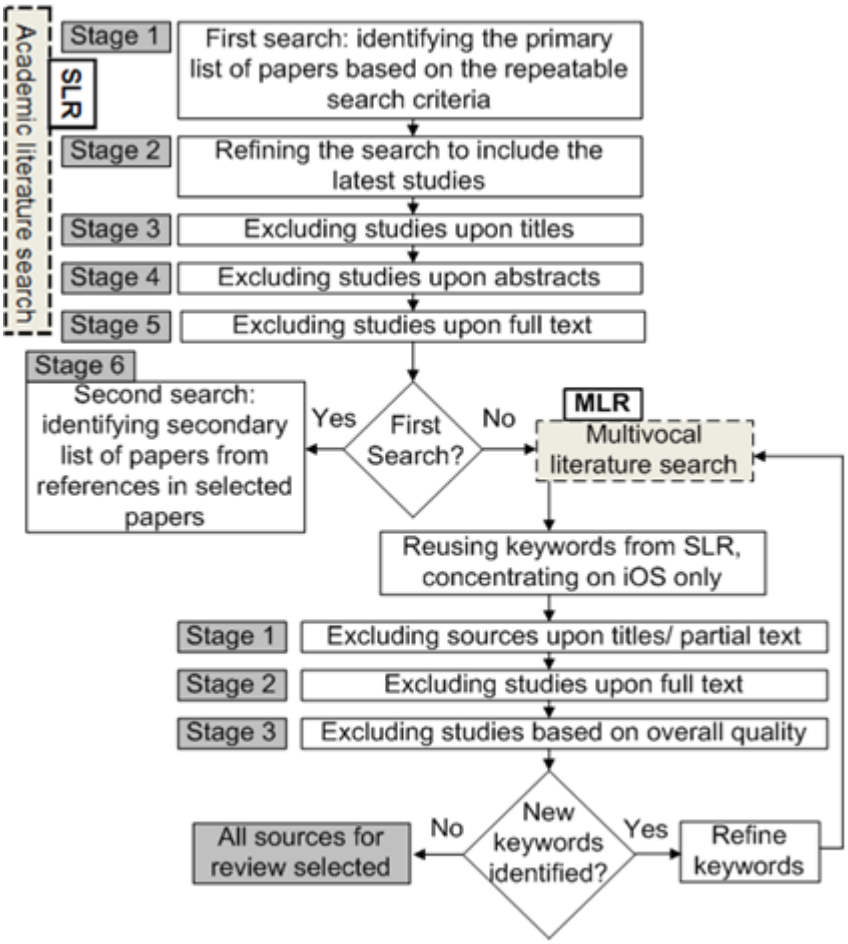


Fig. 2.1. Process of sources selection for SLR and MLR.

The procedure described by Kitchenham and Charters [48] 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 [45]. They define the multivocal sources as accessible, but non-academic writings on the topic.

2.1.2.1. Systematic Literature Review

The search of academic literature for SLR was performed in two iterations. The first iteration was executed using the databases and search criteria that are described below. 12 papers were selected as relevant to answer RQ. The second iteration was executed based on the references in the papers selected after the first iteration. Some relevant sources were found, but they appeared to be non peer-reviewed. The details of each iteration can be found in Table 2.1.

Table 2.1.

Number of Papers Left after Exclusion/ Inclusion during Each SLR Stage

Iteration	Stage	Number of Academic Works after Stage
1	1. Initial repeatable search (duplicates removed)	946
	2. Refined search to include works from 2014	972
	3. Exclusion upon titles	33
	4. Exclusion upon abstract	18
	5. Exclusion upon full text	12
2	6. Secondary search based on references in selected results	0
	Total:	12

1) Databases. The following databases were used to search the keywords described in the Search Keywords section: Springer Links (Springer), IEEE Xplore (IEEE further in the text), and ACM Digital Library (ACM).

2) Search Keywords. Appropriate keywords were searched in metadata. Due to the search engines differences, metadata should be treated as a search within the title, OR abstract, OR keywords for ACM and as a search within the title only for Springer, while IEEE has an option to search within all metadata at once.

Because preliminary search of keywords “iOS application” and “testing” or “iOS application” and “quality” returned small amount of results, the keyword “iOS” was substituted with “mobile”. It was also given a try to shorten the word “application” to “app”. The following search string was used: (("iOS apps" OR "iOS applications") OR ("iPhone OS apps" OR "iPhone OS applications") OR ("mobile apps" OR "mobile applications")) AND ("quality" OR "testing" OR “verification” OR “validation”).

3) Inclusion/ Exclusion Criteria. Only peer-reviewed papers in English were selected. There was no limitation given on the type of the source (i.e. journals, conference proceedings, etc.). Papers starting from the year 2007 were chosen, because it is the year when iOS (iPhone OS at that time) was released. The year 2013 was chosen as the last year of publication for the

search results repeatability. The search was also refined by adding papers from the year 2014 in order not to miss the latest available information.

Irrelevant papers were excluded upon title, then upon abstract, and then upon full text. The main credit was given to the papers that offered some categorization or general overview of mobile applications testing. Papers that mention only the specific testing type of mobile applications (i.e. unit testing, security, usability, etc.) or that are related to test automation were excluded from the results after additional acquaintance with abstract because they do not focus on the aspects asked in RQ. Only non-shortened version of papers were included if two versions of the same paper for different occasions (e.g. conference proceedings and magazine) were identified.

4) Data Extraction and Synthesis. The data extraction phase involved the extraction of aspects and categories of aspects related to RQ from the selected studies. The categories of multiple non-overlapping aspects are mentioned in some papers, while the detailed description of aspects from single category is mentioned in others. The data synthesis phase includes the merge of aspects from the different papers that appeared to have the same meaning. In order to make the data more usable the aspects were divided between 4 large clusters: Environment, Application Lifecycle, Inside the Application, and (functional or performance aspects of) UI/ UX.

2.1.2.2. Multivocal Literature Review

1) Data Sources and Search Strategy. Sources for MLR were searched in Google (`http://www.google.com/`). The combination of the same keywords as for SLR, excluding the "mobile applications" OR "mobile apps" part, was used for the first search iteration. The keyword "checklist" was added for the second iteration. The first 50 relevant articles per iteration (see Appendix B) based on the Google ranking algorithm were taken for subsequent analysis.

2) Inclusion/ Exclusion Criteria. The sources were excluded during three stages by evaluating 1) Title/ partial text available in the search results; 2) full text; 3) overall quality. The sources related to iOS testing only were included into the final results, i.e. the sources containing only information about general mobile testing aspects were excluded. The sources on security or unit testing, as well as the sources on testing automation were excluded as well. Duplicates were excluded upon title during the first exclusion stage. Some sources were excluded on the second stage because they directly referred to other sources found. The inclusion/ exclusion progress is depicted in Table 2.2.

Table 2.2.

Number of Papers Left after Exclusion during Each MLR Stage

Iteration	Initial	Stage 1	Stage 2	Stage 3
Iteration 1	50	20	5	5
Iteration 2	50	20	5	4
Total				9

3) Data Extraction and Synthesis. The data extraction phase involved the extraction of aspects and aspects categories asked in RQ. Some sources already contain categorized lists of aspects while others are the materials written in narrative. The data synthesis phase includes the merge of aspects from the selected sources. The identified aspects were divided between the same clusters as done for SLR.

2.1.3. Results

2.1.3.1. Summary of Reviews

Despite the fact that the search criteria for SLR includes studies starting from 2007, the first selected study was published in 2009 [41], but the most productive years are 2012 (five studies: [44], [49], [50], [51], and [40]) and 2013 (three studies: [52], [53], and [54]). Two studies [43] and [55] were published in 2011, and one study [42] was published in 2014. Two studies [44] and [43] are related to narrow topic of mobile application lifecycle, one study [54] is related to user complaints about iOS applications, and other nine sources, [41], [49], [50], [51], [40], [52], [53], [55], and [42] are related to the general testing of mobile applications.

Between the sources selected through MLR, seven sources [56], [57], [58], [59], [60], [61], and [62] were published in 2013, and one source was published in 2012 [63] and one in 2014 [64]. Five sources [57], [58], [59], [63], [64] are blog posts, two sources [56] and [62] are testing checklists, one source [60] is a white paper, and one source [61] 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.

2.1.3.2. Aspects of iOS Applications Testing

The aspects that influence the testing of iOS applications gathered through SLR and MLR are shown in Table 2.3. 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 [56], [60], [61], and [62] that have different screen size, resolution & pixel ratio, processing efficiency, memory, and storage capacity, as per [41], [50], [51], [40], [53], [54], [55], [42], [60], [61], and [62]. It is claimed in [41] that functionalities, usability issues in the interface design, and user behavior "to be tested in emulator", while other sources [50], [53], [54], [55], and [64] 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 [49], [52] like headphones [49], [52], [62] and keyboard [49], [52] that can be connected to the device.

It is claimed in many sources [50], [40], [53], [42], [60], [61], and [62] that the variety of operating systems (OS) is an important testing aspect, while OS upgrade is mentioned explicitly only in [53]. 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 [56], [59], [61].

Mobile devices have limited power, processing, and memory resource [41], [44], [49], [51], [40], [53], [54]. Thus resources consumption efficiency plays an important role in application success [41], [49], [51], [40], [54], [56]. Applications should also be checked on different networks, i.e. strong WiFi connection, cellular network (LTE, 3G, EDGE), and in Airplane mode [41], [49], [40], [42], [58], [59], [61], [62]. Different network conditions (e.g. slow connection, packets loss, etc.) should be taken into consideration as well [61]. Different regional settings, like data and time formats [61], [62], as well as time zones [62] 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 [51], [56], [62], work in foreground, stay in background [44], [49], [43], [56], [61], receive memory warnings [44], [49], [43], [55], [62], be interrupted by a call or SMS [52], [56], system alert [52], push notification [56], [61], GPS signal [52], or audio/ video from another application [56], [61], [62]. It can even crash [53], [54], [60], [61]. Or it can also be updated to the next version [53], [61], [62].

[59] warns about the need to check an extended (Asian) on-screen keyboard, while [41] mentions on-screen keyboard as a generic aspect that should be taken into consideration. According to [56] and [61] data can be shared via email or Bluetooth, or another network between the applications. According to [61] and [62] it is necessary to check application's logging and analytics features. Testing of In-App Purchase component is mentioned in [61]. Testing of Web View component is mentioned both in [42] and [61].

An application can be manipulated with a variety of gestures [42], [61]. When animated transitions occur, they must run smoothly [54], [61] irrespectively of the task executed in

parallel. Testing for half pixels glitches and testing of Pull to Refresh feature are mentioned in [61]. The necessity of checking the application both in portrait and landscape is noticed in [41], [49], [61], and [62]. The importance of localization testing is mentioned in [53] and [62]. [56] 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 [57], [60], [61], [62], [63].

Table 2.3

Aspects of iOS Applications Testing

Environment		
Hardware		
Devices	iPad, iPhone, iPod [56], [60], [61], [62].	Screen size, resolution & pixel ratio, processing efficiency, memory, storage capacity; [41], [50], [51], [40], [53], [54], [55], [42], [60], [61], [62] motion activities [61], [62].
Simulator	[41], [50], [40], [53], [55], [64].	
External Accessories	Headphones [49], [52], [62], keyboard [49], [52]; wired/ wireless [49], [52].	
Operating System		
OS Variety	[50], [40], [53] [42], [60], [61], [62].	OS upgrade [53].
Restrictions and Privacy Settings	[41], [53], [54], [42]. Safari, Camera, Siri, IAP (in-app purchase), Location Services, Contacts, Calendars, Photos, Social Networking, Microphone, Motion Activities, Cellular Data Use, Background App Refresh. [56], [59], [61]	
Resources		
Limitations	Lack of storage, amount of memory, running out of battery, processing capabilities. [41], [44], [49], [51], [40], [53], [54]	
Consumption	Memory consumption, battery consumption. [41], [49], [51], [40], [54], [56]	
Connectivity		
Network Types	WiFi, Cellular networks; [41], [49], [40], [42], [58], [62] Bluetooth [41], [49], [40], [62]; Airplane mode. [61], [62]	
Network Conditions	[41], [49], [51], [40], [52], [54]. Strong/ no/ poor connection; connection loss. [56], [58], [59], [61] Ask for connection [41].	
Internalization		
Region Formats	[51]. Date format, hour format [61], [62]	
Date/ Time Settings	Switching between time zones, system time too fast/ too slow. [61]	

Aspects of iOS Applications Testing (continued)

Application Lifecycle	
Installing and Launching	[51], [56], [62].
Background	[44], [49], [43], [56], [61].
Crash	[53], [54], [60], [61].
Low-Memory Warnings	[44], [49], [43], [55], [61]
Interruptions	[41], [44], [49], [51], [43]. Call/ SMS [52], [56]; push notifications [56], [61], system alerts [52]; GPS signal [52]; audio/ video [56], [61], [62].
Application Update	[53], [61], [62].
Inside the Application	
Keyboard	[41]. Extended keyboard [59].
Data Import/ Export	Email; Bluetooth/ network (peer to peer) [56], [61].
Logging/ Analytics	[61], [62].
In-App Purchases	[61].
Web View	[42], [61].
UI/ UX	
Gestures	[42], [61].
Smooth Animation	[54], [61].
Pull to Refresh	[61].
Orientation	Portrait, landscape. [41], [49], [61], [62]
Half Pixels	[61].
Localization	[53], [62]. Native characters and special symbols [56].
Accessibility	VoiceOver, accessibility zoom, etc. [57], [60], [61], [62], [63]

2.1.4. Discussion and Implications

Despite the fact that the Results section shows the identified aspects of iOS applications testing gathered through SLR and MLR, the author feels the necessity to discuss the details of identified aspects. There are also some aspects that are known to the author (like iAd, update of Xcode, AirDrop, etc.), but they are missing in the reviewed literature. Some of the details are provided in the reviewed sources. Others are added based on the author's more than five years of professional experience of leading more than 20 iOS applications testing projects for several Global Fortune 500[2] and other multinational companies, giving the references to iOS Developer Library[3] or other credible sources where possible.

[2] http://money.cnn.com/magazines/fortune/global500/index.html
[3] https://developer.apple.com/library/

2.1.4.1. Hardware

1) Devices. While there are four types of iOS devices, business applications are mostly developed for iPads[4], and sometimes have reduced iPhone versions[5]. iPods and Apple Watch devices generally are out of scope, however Apple Watch begins to receive more and more attention. The variety of iOS device is depicted in Fig. 2.2.

Fig. 2.2. iOS Devices Variety.[6]

iPad 1st generation devices, as well as iPhone 3G, iPhone 3GS, and iPhone 4 are not taken into consideration anymore when new applications for iOS are developed. Only iPhone 4 supports iOS 7 (the latest iOS version at the moment of writing is iOS 9), but three other mentioned devices already are not[7].

iPad 2[8] and iPad mini[9] both have non retina display (i.e. a display with lower pixel density than the latest iOS devices) and generally the same hardware options. They are the least powerful iPad devices that support the latest iOS version. Special checks that application design fits the small screen of the device and that every UI control can be easily interacted with should be performed on iPad mini.

[4] http://www.apple.com/ipad/business/
[5] http://www.apple.com/iphone/business/
[6] http://gbtimes.com/world/apple-iwatch-and-other-smart-watch-competitorstime-savers-or-waste-time
[7] http://support.apple.com/kb/ht5457
[8] http://support.apple.com/kb/sp622
[9] http://support.apple.com/kb/SP661

iPad 4[10] and iPad 3[11] both have retina displays, but iPad 4 is more powerful than iPad 3. Generally, it is enough to have only one device of any generation to cover this category of devices.

iPad Air[12] and iPad mini retina[13] both have faster GPU (but still retina display) and M7 64-bit core processor that has built-in hardware for motion activities like *accelerometer*, *gyroscope*, and *compass.*

iPad Air 2[14] and iPad mini 3[15] are equipped with Touch ID[16] technology. iPad Air is also equipped with even faster GPU and M8 64-bit core processor that has a barometer sensor in addition. iPad Pro[17] is an iPad with the largest screen size and more powerful M9 processor.

iPhone 4[18] and higher all have retina displays. iPhone 4S[19] has a faster dual core processor in comparison with iPhone 4. iPhone 5[20] and iPhone 5C[21] are both packed with even faster next generation processor. iPhone 5S[22] is packed with already mentioned M7 64-bit core processor and fingerprint identity sensor.

Despite the fact that iPhone 5th generation devices have larger screen size in comparison with iPhone 4th generation devices, applications designed for iPhone 4th generation devices can still run on iPhone 5th generation devices, but there are black bars above and below application content, *unless* properly named and to a larger screen accordingly sized launch image is provided.[23]

iPhone 6[24] and iPhone 6 Plus[25] have M8 64-bit core processor, larger and even larger screen size, and already mentioned fingerprint identity sensor, barometer, and other sensors. Separate image resources should be prepared for applications to look smooth on iPhone 6 Plus. The "S" upgrade of iPhone 6th line[26, 27] comes with yet more powerful M9 processor and 3D Touch[28] technology that recognizes the power of the pressure of the touch. This technology brings two new gestures like Peek and Pop.

[10] http://support.apple.com/kb/sp662
[11] http://support.apple.com/kb/sp647
[12] http://support.apple.com/kb/SP692
[13] http://support.apple.com/kb/SP693
[14] http://support.apple.com/kb/SP708
[15] http://support.apple.com/kb/SP709
[16] http://support.apple.com/en-us/HT5883
[17] https://support.apple.com/kb/SP723
[18] http://support.apple.com/kb/sp587
[19] http://support.apple.com/kb/sp643
[20] http://support.apple.com/kb/sp655
[21] http://support.apple.com/kb/SP684
[22] http://support.apple.com/kb/SP685
[23] https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AdvancedAppTricks/AdvancedAppTricks.html
[24] http://support.apple.com/kb/SP705
[25] http://support.apple.com/kb/SP706
[26] https://support.apple.com/kb/SP726
[27] https://support.apple.com/kb/SP727
[28] http://www.apple.com/iphone-6s/3d-touch/

iPhone 6th line supports Near Field Communications-based mobile payment technology Apple Pay[29]. Apple Pay can be used also on iPhone 5th line, but only when paired with Apple Watch. Payments can be made also on iPad Air 2, iPad mini 3, and iPad Pro, but only within the applications.

Generally speaking, one device from each generation would be enough to cover the whole set of iPhones, in case of application under test does not rely on the specific function of the device like motion activity or finger print of iPhone 5S and higher or Siri (advanced voice control) that is available only starting from iPhone 4S.

2) Device vs. Simulator. The author's professional experience supports the statement expressed in [50], [53], [54], [55], and [64] that for achieving good quality of the application, it should be tested on the device rather than on the simulator, because testing results can vary. It also should be taken into consideration that application can behave differently when it is built in debug, not in release mode.[30]

3) External Accessories. There are different kinds of accessories, both wired and wireless [49], [52], that can be attached to the device: headphones [49], [52], [62], keyboard, [49], [52], stylus, etc. It can occur that an application handles the inputs and outputs from/ to external accessories in a different way than it does without them, or it does not handle them at all.[31] External accessories from different manufacturers can behave differently, e.g. styluses from different manufacturers can have different configurations inside the application in order to handle the palm (interaction) rejection, etc.[32]

2.1.4.2. Operating System

1) iOS Variety. Release of the new iOS version almost always leads to the major retesting cycle for the non-trivial applications. New Xcode[33] version (that includes new version of iOS SDK and compiler)[34] is shipped together with the new iOS version. Thus, there can be completely different test results when the same code is built by the different Xcode versions.

The following update strategy is followed by the development organizations which the author works or worked for when the new iOS version is released:

1) Current application version built by previous the Xcode is checked on the new iOS version (preliminary checks are done already on Beta or GM versions).

[29] https://www.apple.com/apple-pay/
[30] https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/DebugYourApp/DebugYourApp.html
[31] https://developer.apple.com/library/ios/featuredarticles/ExternalAccessoryPT/Articles/MonitoringEvents.html
[32] http://www.apartmenttherapy.com/tablet-stylus-test-lab-comparison-of-pencil-intuos-pogo-connect-jot-script-tech-test-lab-reviews-196850
[33] https://developer.apple.com/xcode/
[34] https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/WhatsNewXcode/00-Introduction/Introduction.html

2) Major failures, if any, are fixed, and the application is released with remark that it supports the latest iOS version.
3) More thorough testing cycle follows when the application current version is built by the new Xcode afterwards.

It is possible to leave the application version built by the previous Xcode for some period of time if, for example, active development currently is not planned. But here is a list of situations when developers are forced to rebuild the application with the new version of Xcode:

- New iOS version does not support the methods that were previously deprecated, but still used in the application; new supported methods that substitute the deprecated ones are available with the new Xcode, e.g. detection of UDID.[35]
- Apple announces that all the new applications or application updates submitted to the App Store must be optimized for new iOS and built with the latest Xcode.[36]
- Application should be redesigned for the marketing purposes, because of the iOS redesign (as it occurred with iOS 7[37]), but new UI is achieved using the latest Xcode only.

It is worth mentioning that devices with the previous iOS version should always be available and handled carefully in case some of the applications developed within the organization still support it. It should not be forgotten that there is no official way to install any previous major iOS version after the release of the latest major iOS version[38]. It should be taken into consideration that not all users update iOS version as soon as it is released[39], but can continue to use the previous one for quite a long period of time. From the author's experience, it is especially applicable for enterprise users – they update iOS version only after the enterprise infrastructure that supports the latest iOS version is ready.

2) Restrictions and Privacy Settings. In iOS a user can set different restrictions, both system and application wise, on the usage of different hardware or OEM software. For example, it is possible to restrict the usage of Safari, Camera, Siri, IAP (In-App Purchase), Location Services, Contacts, Calendars, Photos, Social Networking, Microphone, Motion Activities, Cellular Data Use, Background App Refresh, etc. [56], [59], [61] The application should handle cases when it tries to access the restricted item. The user should also be warned about the

[35] https://developer.apple.com/library/ios/documentation/uikit/reference/UIDevice_Class/DeprecationAppendix/AppendixADeprecatedAPI.html
[36] https://developer.apple.com/news/?id=12172013a
[37] http://www.apple.com/pr/library/2013/09/10iOS-7-With-Completely-Redesigned-User-Interface-Great-New-Features-Available-September-18.html
[38] http://www.itproportal.com/2013/09/29/why-apple-wont-allow-you-to-downgrade-your-iphone-from-ios-7-to-ios-6/
[39] http://appleinsider.com/articles/13/12/31/ios-7-now-installed-on-78-of-active-apple-handheld-devices

restriction and instructed how to remove it[40] or offered to remove the restriction within the application if it is possible.

2.1.4.3. Resources

1) Limitations and Consumption. Due to the fact that a mobile device has more limited storage, memory, power, and processing capabilities than an ordinary PC [41], [44], [49], [51], [40], [53], [54], examination of how the applications handle these limits and operate within these limits are of special interest. The application should check for the free storage availability when the new data is added/ downloaded. Otherwise, from the author's experience, the user will not be able to operate with the data that already is inside the application due to crashes. The application should be checked for efficient battery consumption as well [41], [49], [51], [40], [54], [56]. It can be verified using Xcode Instruments tools[41]. Battery usage logging can also be enabled on the device that is provisioned for the development[42]. Instruments tools can also be used for profiling the efficiency of memory and processor resource utilization.

2.1.4.4. Connectivity

1) Network Types. During Alpha testing the application is mostly checked in the laboratory environment [41]: on the strong WiFi connection and in the Airplane mode. The working on the cellular data (LTE, 3G, EDGE) should be checked as well [41], [49], [40], [42], [58], [62], especially if the application utilizes a lot of traffic. The user, at least, should be warned when large data synchronization occurs on the cellular network.

2) Network Conditions. There are different network conditions possible [41], [49], [51], [40], [52], [54], [56], [58], [59], [61] (e.g. slow connection, packets loss, etc.). It should be checked if under these conditions:

- The application handles different network conditions on the first launch. [61]
- Proper error messages are shown on timeouts and other network errors. [61]
- The interaction with UI (i.e. the main thread) is not blocked. [61]
- The corrupted data is not stored, or at least can be redownloaded.

For simulating different network conditions Apple Network Link Conditioner can be used [M45]. This tool is a part of Xcode Developer Tools[43] and can simulate network conditions on

[40] http://support.apple.com/kb/ht4213
[41] https://developer.apple.com/library/ios/documentation/AnalysisTools/Reference/Instruments_User_Reference/Introduction/Introduction.html
[42] https://developer.apple.com/library/ios/recipes/xcode_help-devices_organizer/articles/provision_device_for_development-generic.html
[43] https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/WhyNetworkingIsHard/WhyNetworkingIsHard.html

the device if the network connection from Mac is shared. It can also be enabled directly on the device that is provisioned for development.

Sometimes it is also necessary to check a poor connection or a connection loss/ switching in the real world. From the author's experience, the most common cases that should not be simulated, but should be checked in the field are:

- The traffic loss while the device "thinks" that it is still connected to the network (e.g. entering the elevator or walking outside the network coverage).
- Switching from WiFi to the cellular network and vice versa, switching from one WiFi access point to another, switching between different cellular network types.
- Only cellular network conditions (e.g. inbound/ outbound connection speed, packet loss ratio, etc.) can be simulated, but the device will still think that it is on WiFi. Thus, the real cellular network should be used to check the cellular network specific functionality of the application.
- The situation when the device is not connected to any network should be checked separately to make sure that this condition is treated the same way as the Airplane mode.

2.1.4.5. Internalization

1) Region Formats. Applications should be tested using different region formats [51] that have different hour format (24 or 12) [61], [62] and different coma separators. For example, German Switzerland and United States regions cover these both differences. From the author's experience, the specific Arabic and Israel region formats should be explicitly tested if the application's functionality is directly related to the calendar and weekend days.

2) Date/ Time Settings. When the application receives updates from backend, and especially when creation/ update timestamps for items are visible (but the same also applies for locally created items), it is necessary to check how the application behaves with different time settings [61]:

- When switching between time zones.
- When the system time is too fast or too slow.

Besides checking that functionality works properly itself, it is necessary to check that relative times are properly calculated [61].

2.1.4.6. Application Lifecycle

1) Installing and Launching. The application should be installed both on the device that already contained some version of the application and on the clean device after the factory reset. The user should be warned through a message or a progress bar in case the access to the application functionality is given in more than 5 seconds after launching. [56]

2) Background. The background mode is one of the major cycles of iOS application lifecycle. If the application cannot be sent to the background in approximately 5 seconds, then iOS kills it. The same is applicable when going back to the foreground. [44] That is why it is necessary to check that the application changes the state in sufficient amount of time even with the large amounts of data inside. The application can also perform the refreshes in the background using the special multitasking feature provided in iOS 7 or if it uses the Location Services, or plays audio content in background, etc. It should be checked that all the data is preserved [62], but specific data is updated and is not corrupted after the application is returned to the foreground. All the animations should be restarted as well – it does not occur automatically.

3) Locked Device. Apple warns that improper design or implementation of cryptographic operations can introduce performance or battery life problems. Locking the device with passcode can influence the applications that can operate in background. What is more, the device denies the access to the keychain and files.[45] From the author's experience, the incidents including data loss and crashes can occur if the application needs access to the keychain during the background activity, but the situation when the keychain is not available is not handled properly. It usually takes a long time to isolate the cause of such incidents. It is easy to crash the application just by frequent locking with a passcode and unlocking the device if the file data protection strategy is poorly designed.

4) Crashes. There is an option to use crash reports [61] when the tester cannot reproduce the exact steps that led to the crash. Some crash reports if symbolicated (i.e. converted to the proper stack trace using debug symbols of the build)[46] can give a hint on the exact scenario that led to crash. Others are not useful if the crash occurred not in the application, but in iOS itself.

5) Low-Memory Warning. When iOS needs more memory, it unloads applications that are currently in the background [49], [43], [61]. Prior to iOS 6, if the application needed even more memory it could unload cashed images (if cashing was performed) and not visible views of the

[44] https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html
[45] https://www.apple.com/ipad/business/docs/iOS_Security_Oct12.pdf
[46] https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AnalyzingCrashReports/AnalyzingCrashReports.html

currently running application. In such cases it was possible to see only the placeholders of images or the application could even crash if unloaded data reload was not properly handled during the further navigation activities. Now developers must handle actions to perform when memory warning is received completely on their own.[47, 48] If the application utilizes a lot of memory (usually it means that there are memory leaks in the application) then it can be fully unloaded from the device memory by iOS itself.[48] Low-memory warnings can be simulated by Xcode Instruments (but only for the simulator).[48, 49] From the author's experience, they can be easily reproduced on the device when many heavy pages are loaded in Safari or when photos or videos are made while the application under test is running on the background. Working with very large data or quick and frequent refreshes of data in UI collections can cause low-memory warnings when the application under test is running on the foreground.

6) Interruptions. The application should preserve its state and should not freeze if it receives an incoming call or SMS [52], [56], system alert [52], or local, or push [56], [61] notification while being in the foreground, especially when activities occur on the main thread.

It is possible to open the application through the push notification if it is received when the application is in the background or closed. Different navigation start points should be checked in case the application also does some navigation actions inside itself on confirming the push notification. The application icon badge update should also be checked including the case when several updates are received in a row.[50, 51]

For applications that play audio/video it should be checked that other audio/ video streams are paused on in-application audio stream start. It should be checked if audio continues to play or not when the application is in the background (to play or not - it depends on the requirements). [56], [61], [62] It is worth mentioning that audio/ video inside the Web Views is handled in a different way than audio/ video played natively.[52]

7) Application Update. The migration process of the application from the previous versions should be tested before the new version of the application that will be available to the final user is released. [61], [62] After the application is updated from the previous version it should be checked that:

- The data is not corrupted.[53]
- The user preferences stay in place. [61]

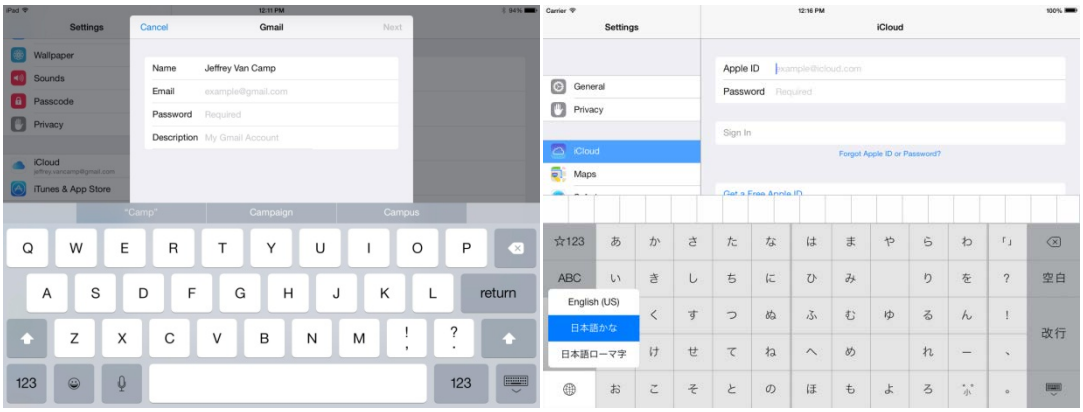
[47] https://developer.apple.com/library/ios/documentation/uikit/reference/UIViewController_Class/Reference/Reference.html
[48] https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/PerformanceTuning/PerformanceTuning.html
[49] https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/InteractingwiththeiOSSimulator/InteractingwiththeiOSSimulator.html
[50] https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Introduction.html
[51] https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/IPhoneOSClientImp.html
[52] https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/00_Introduction.html
[53] https://developer.apple.com/library/ios/technotes/tn2285/_index.html

- Saved credentials are still there. [61]
- Previously registered push notifications are still received.[53] [61]

The updates should be performed using the different possible paths starting from the very first application release [61]. From the author's experience, in some cases (e.g. due to the requirements change, or the incomplete data model design during the first release) data model changes can be so significant that users are asked to perform the backup of their data and to perform the clean install of the application. Encrypted data migration is also the subject of interest. When there is a backend server and it is updated as well, it is necessary to check the old application versions on the new server version if there is no mechanism that does not allow connecting to the server with the old versions of the application.

2.1.4.7. Inside the Application

1) Keyboard. Editable UI elements should be focused through auto scroll after onscreen keyboard appears. In practice, it is often forgotten to check how they behave in case of split, undocked, extended [59] (see Fig. 2.3) or external keyboard [49], [52], only docked and merged onscreen keyboard is taken into account. From the author's experience, non standard keyboard appearances often influence the usability of those editable elements that are placed near the screen bottom border.



a) Basic keyboard. b) Extended keyboard.

Fig. 2.3. The difference between basic and extended keyboards.[54,55]

[54] http://www.digitaltrends.com/apple/ios-8-review/
[55] http://stackoverflow.com/questions/19626619/where-is-the-ios7-simulator-japanese-keyboard-dictionary-located

2) Data Import/ Export. Many applications support different file formats that they can operate with. There are different ways how supported file formats can be imported into or exported from the application. They are:

- Open In from email, web browser, or other applications;[56]
- via Air Drop;[57]
- via Email; [56], [61]
- via iTunes;[58]
- via Photos application/ Camera;[59]
- via Bluetooth/ network (peer to peer); [56], [61]
- import (download and open) from URL.[60]

It should be checked that the application handles (i.e. is registered to open and can open[61]) supported file formats in non case sensitive manner.[62] Names of the exported files should be verified as well.

Files can be sent via email from the application. In most cases, iOS native email client is used for this purpose. It should be checked that there are default to, subject, and body set on email creation. The application should also properly handle the case when there is no email account configured. [61]

From the author's experience, there are not many problems encountered when images are imported from the Photos application using the native view controller. But in case when custom view controller is used, it should be more strictly checked how it is synchronized with Photos application. The robustness of the Camera component usage is also the subject of worries. The Camera component tests should include the device orientation change, rotation lock, background, etc., i.e. the same aspects that should be checked for every mobile application.

3) Logging/ Analytics. Public analytics engines are often used for collecting crash reports, feature usage statistics and other logs for further development activities and testing thoroughness prioritization [61], [62]. Analytics is mostly used for publically available applications without own backend server. Based on the author's experience, if analytics is used then the main points that should be checked are:

[56] https://developer.apple.com/library/ios/qa/qa1587/_index.html
[57] http://support.apple.com/kb/ht5887
[58] http://www.apple.com/itunes/
[59] https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/CameraAndPhotoLib_TopicsForIOS/Introduction/Introduction.html
[60] https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/URLLoadingSystem/Tasks/UsingNSURLDownload.html
[61] https://developer.apple.com/library/ios/documentation/filemanagement/conceptual/documentinteraction_topicsforios/Introduction/Introduction.html
[62] https://developer.apple.com/library/ios/qa/qa1697/_index.html

- Analytics gathering should handle situations when the data is not available or has another format than expected. It is better to send the wrong one or no statistics than to break the UX.
- The statistics should not be sent via cellular networks. In most cases only WiFi connection should be used.
- The analytics should not gather the data about the user without his/ her permission. The user should be warned about how and where the data will be used.[63]
- Collecting the data should not break the UX in any other way.

Enterprise applications can have other, more strict and extensive rules for logging depending on the corporate policy. Own logging protocols are used in such cases.

4) In-App Purchase. In-App Purchase (IAP) is a business model that allows the user to buy virtual or digital consumables, non-consumables, and subscriptions within the application that is distributed via Apple App Store. It should be checked that the purchased items are available on all the devices that are registered for the particular user, and that purchases are restored after the application reinstall, clean install, and iOS update or clean install.[64]

IAP products can be tested using special test users on Apple test environments. It is also possible to test auto-renewable subscriptions on these environments, because they have compressed durations for testing purposes.[65]

IAP password cashing system setting is of the special interest. The password can be saved for 15 minutes or asked each time the user makes any IAP.[66] The application should be checked for handling both options. [61]

5) iAd. iAd is Apple's platform that allows to "generate revenue and promote … apps" by showing an advertisement within the applications.[67] Test advertisements, including the erroneous one can be sent "over local networks or USB using iAd Producer, or over the carrier network using Apple's test servers".[68] There are two types of advertisement available: banner views and full-screen advertisements.[69] Apple suggests checking that the application shows only fully loaded advertisements. The application should pause other activities when the user begins the interaction with a banner and should restart them when the user finishes (or system cancels) the interaction with a banner. Advertisements should appear quickly and response to the device orientation changes.[69]

[63] https://developer.apple.com/appstore/resources/approval/guidelines.html
[64] https://developer.apple.com/in-app-purchase/In-App-Purchase-Guidelines.pdf
[65] https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnectInAppPurchase_Guide/Chapters/TestingInAppPurchases.html
[66] http://support.apple.com/kb/ht6088
[67] https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/iAd_Guide/Introduction/Introduction.html
[68] http://support.apple.com/kb/HT5245
[69] https://developer.apple.com/library/ios/documentation/userexperience/conceptual/iAd_Guide/TestingiAdApplications/TestingiAdApplications.html

6) Web View. Web View is a part of WebKit. Web Views are used to represent the web content inside the native mobile applications.[70] The native application is called a hybrid when most of the data inside it is represented using Web Views.[71] Web Views are often used in order to open different file formats[72] or to login to the different content providers. It should be checked that the links inside the Web Views are opened in the way they are designed to (they are opened in the same view by default[73], but they often should be opened in a default browser, for example). From the author's experience, unnecessary scroll bars and bouncing effects should be eliminated if any.

2.1.4.8. (Functional Aspects of) UI/ UX

1) Gestures. The application can be manipulated with a variety of gestures like tap, double tap, touch and hold, pinch, pan, swipe, etc. It should be checked that gestures bring the same user experience as suggested in iOS Human Interface Guidelines.[74] The applications made by Apple can be used for reference. Based on the author's practice, if some elements on the definite application screen support non-trivial gestures other than single tap, other screen elements around should be checked for interaction using the same non-trivial gestures. It also should be verified that unexpected interactions with multiple UI elements at once are not allowed, because such actions often lead to crash.

It is worth mentioning that minimal suggested tappable area is 44 x 44 pt.[75]

2) Smooth Animation. The animation is used to improve UX when the application responds to the user actions or when it provides the user with a feedback about the occurring on the screen. But they should not be "excessive or gratuitous" otherwise they "can obstruct app flow, decrease performance, and distract users from their task".[76] The animation should be smooth irrespectively of the currently running background tasks, thus the author recommends testing different animated transitions for smoothness while heavy background tasks occur.

3) Pull to Refresh. Pull to Refresh [61] feature is a very common user experience mechanism that is used for performing delta data loads in mobile applications (see Fig. 2.4).[77] It should be verified that Pull to Refresh mechanism loads only the new data, not the whole available data set, and that already loaded data is persisted in case of the Pull to Refresh update

[70] https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/DisplayWebContent/DisplayWebContent.html
[71] http://blogs.telerik.com/appbuilder/posts/12-06-14/what-is-a-hybrid-mobile-app-
[72] https://developer.apple.com/library/IOs/qa/qa1630/_index.html
[73] https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/DisplayWebContent/Tasks/SimpleBrowsing.html
[74] https://developer.apple.com/library/ios/documentation/userexperience/conceptual/MobileHIG/InteractivityInput.html
[75] https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/LayoutandAppearance.html
[76] https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/Animation.html
[77] http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1
&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=8,448,084.PN.&OS=PN/8,448,084&RS=PN/8,448,084

failed. It also should be checked how it behaves when the current data/ time and/ or data settings (e.g. format and zone) are changed.

Fig. 2.4. Pull to Refresh Example.[78]

4) Orientation. The application should be checked in both orientations if applicable [41], [49], [61], [62].[79] Based on the author's experience, it can occur that UI elements are wrongly placed on the orientation change, and the application can crash when the user interacts with misplaced elements (it often occurs with popovers[80]). The application can also crash when the device is rotated during the execution of heavy operations. Executing the actions after the rotation with the rotation lock option enabled is also the subject of interest, because there are several ways how the device orientation can be checked and how the device orientation change is detected by the application.[79, 81]

5) Half Pixels. Sometimes there are half-pixels [61] and other unexpected blurs[82] noticed when using the application. They occur when UI elements are scaled or when their size and origin are calculated, but not rounded to the whole pixels. The same applies to the fonts. These UI glitches are more visible on the non-retina displays and are often inspected in practice using the 3-fingers accessibility zoom[83].

6) Localization. The following should be checked in case the application supports localizations:

- Localized text in images. [61]
- Localized translated text fits the available area. [61]

[78] http://www.tapsmart.com/tips-and-tricks/pullrefresh/
[79] https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/RespondingtoDeviceOrientationChanges/RespondingtoDeviceOrientationChanges.html
[80] https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIPopoverController_class/Reference/Reference.html
[81] https://developer.apple.com/library/ios/qa/qa1688/_index.html
[82] https://developer.apple.com/library/mac/documentation/userexperience/conceptual/applehiguidelines/IconsImages/IconsImages.html
[83] http://support.apple.com/kb/HT5018

- The same text is localized in exactly the same way when used in different parts of the application.
- Right-to-left text input and alignment [61] for Arabic and Hebrew languages.
- Native and special characters:
    - persistence in a database or a file;
    - printing and display [56];
    - writing to log;
    - handling both by the client and the server.

7) Accessibility. There are plenty of accessibility features available in iOS [57], [60], [61], [62], [63], i.e. VoiceOver, accessibility zoom, bold text, invert colors, etc.[83] They all change the way how the system and the applications look and respond to the gestures. Thus it should be checked that enabling the accessibility features of the system does not break the application.

2.2. Other Mobile Operating Systems

2.2.1. Google Android

The fragmentation differs Android from all other mobile platforms. There are four screen sizes: small, normal, large, and extra-large. Screens can also have different density[84]:

- low-density (*ldpi*) screens (~120dpi);
- medium-density (*mdpi*) screens (~160dpi);
- high-density (*hdpi*) screens (~240dpi);
- extra-high-density (*xhdpi*) screens (~320dpi);
- extra-extra-high-density (*xxhdpi*) screens (~480dpi);
- extra-extra-extra-high-density (*xxxhdpi*) uses (~640dpi) (for launcher icons);
- televisions (*tvdpi*) screens (~213dpi).

There are several representatives in each group of size and density combination. According to the report by OpenSignal [65] there were 24 093 distinct Android devices available on the market at August, 2015.

There are also plenty of (outdated) operating system versions still in use. There are two reasons for this:

- device manufacturers release their own tweaks and skins (launchers) on top of the original Android;
- device manufacturers have to write the updates for drivers.

[84] http://developer.android.com/guide/practices/screens_support.html

While Apple forces the updates, the Android device manufacturers are left to their own to support or not the latest OS updates. That is why users often install custom read-only memory (ROM), like CyanogenMod by themselves that allows using the features from the latest Android versions even if there is no official update from the manufacturer.

While Safari is the main browser on iOS, and all other custom browsers can use the limited functionality of UIWebView from WebKit[85], it is not the case on Android. The top 10 Android browsers are[86]:

- Chrome Browser
- Chrome Beta
- Opera browser for Android
- Opera Mini mobile web browser
- Mozilla Firefox for Android
- Dolphin Browser for Android
- Ghostery Privacy Browser
- Link Bubble Browser
- Puffin Web Browser
- Mercury Browser for Android

The set of browser to test web app on should be chosen based on their popularity on the target market. Many of them use their custom engines for rendering the web content.

2.2.2. Microsoft Windows/ Microsoft Windows Mobile

Windows 10 Mobile runs both on ARM and IA-32 (32-bit x86) processors. This is achieved using Universal Windows Platform (UWP)[87]. There are three main aspects to consider when testing Windows apps:

1) The difference between ARM and x86 processors architecture – some mathematics can work differently.
2) The fragmentation of the screen sizes – as soon as there are lot of vendors doing the mobile devices running on Windows – the target set of the test devices to be chosen accordingly.
3) The split screen feature. It should be checked that app is still usable when it occupies ¼, ½, and 3/4 of the screen.

[85] https://developer.chrome.com/multidevice/ios/overview
[86] http://www.androidcentral.com/10-best-android-browsers
[87] https://msdn.microsoft.com/en-us/windows/uwp/get-started/whats-a-uwp

2.3. 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.

3. MOBILE APPLICATIONS FUNCTIONAL SECURITY TESTING

3.1. Apple iOS

3.1.1. 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. [66]

3.1.2. 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 Charles[88], Fiddler[89], 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.

3.1.3. 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 Xcode[33], or using such third party apps like iFunBox [90]or iPhoneExplorer[91]. 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.

3.1.4. 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

[88] http://www.charlesproxy.com/
[89] http://www.telerik.com/fiddler
[90] http://www.i-funbox.com/
[91] https://www.macroplant.com/iexplorer/

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.

3.1.5. 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 in-app 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. 3.1).
- Entry points for development and test settings are left accessible within run-time of production version.

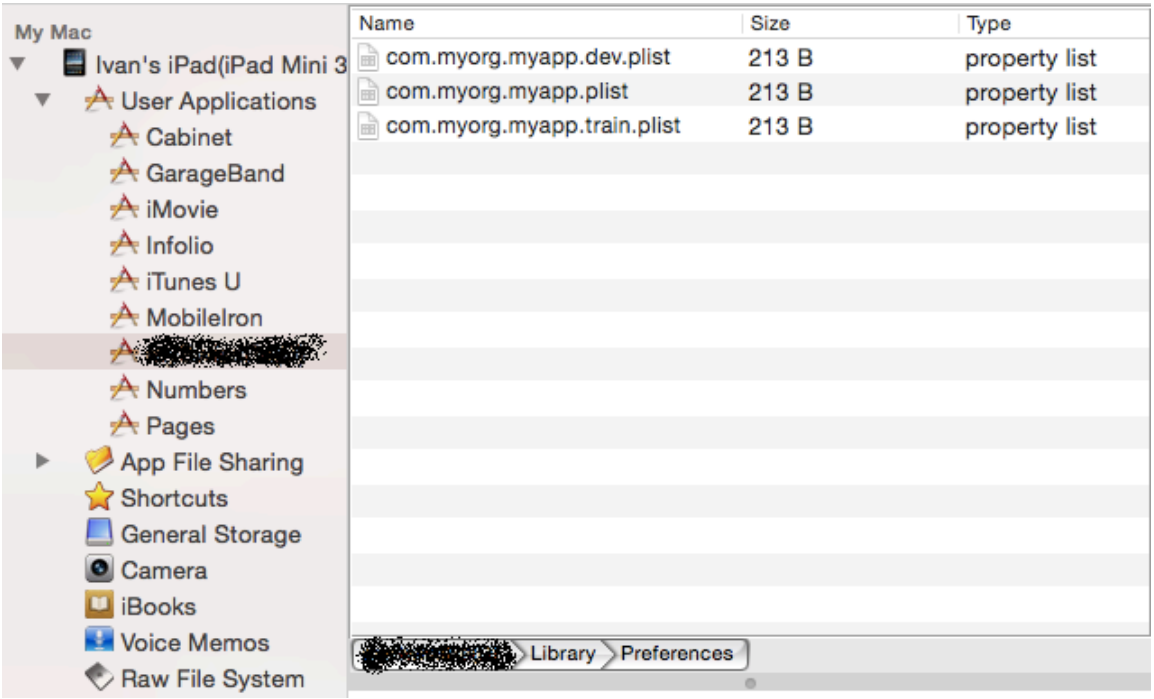


Fig. 3.1. Example of the development and test settings files in production build.

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.

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 Cycript[92], iNalyzer[93], 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.

3.1.6. 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 author's 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.

3.2. Other Mobile Operating Systems

The usage of the secure network protocols, as well as in-app data base encryption are, of course, important for Google Android and Windows mobile apps as well. But, in general, another dimension of security is topic of interest for Google Android – the security of OS itself. Android is considered less secure than iOS that influence on its market share as an enterprise solution, especially in Bring Your Own Device (BYOD) case. "The Myths about and Solutions for an Android OS Controlled and Secure Environment" are studied by the author and colleagues

[92] http://www.cycript.org/
[93] https://appsec-labs.com/inalyzer/

in [67]. The solution to improve the security of a mobile environment in general and Android environment in particular is proposed by the author and colleagues in [68].

The situation of OS security for Windows for mobile is the same as for the Windows on desktop because of UWP technology[87]. That means that to ensure the security of Windows on mobile device the anti-virus software to be installed. But due to the very low market share Windows on mobile is not a target for malware creators so far[94].

[94] http://betanews.com/2015/06/11/windows-phone-security-is-top-notch-says-kaspersky/

4. MOBILE APPLICATIONS TEST AUTOMATION

4.1. 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. [17]

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.

4.2. Solutions for Automated UI Testing of iOS 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-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.

4.2.1. OEM Automation Tools

4.2.1.1. 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. [69] 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. More thorough comparison, as well as study of their pros and cons is described in Chapter 5.

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 TuneupJS[95] 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 ImageMagic[96] 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.js[97] is a CSS-style selector engine for UIAutomation. It also allows accessing UIAElements and executing the commands with a shorter notation.

4.2.1.2. Google Testing Support Library

Google Testing Support Library consists of three main parts[98]:

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

[95] http://www.tuneupjs.org/
[96] http://www.imagemagick.org/
[97] http://www.cozykozy.com/mechanicjs/
[98] http://developer.android.com/tools/testing-support-library/index.html

- 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.

4.2.1.3. Microsoft Coded UI Tests

Coded UI Tests is an analogue for UI test automation for Windows apps[99]. 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#.

4.2.2. API-based Tools

4.2.2.1. Appium

Appium[100] is an open source test automation framework for use with native, hybrid and mobile web apps. It drives iOS and Android apps using the WebDriver protocol. Tests can be written in C#, Java, JavaScript, Perl, php, Python, and Ruby. Native automators (UIAutomation/ UI Automator) are called at the end (see Fig. 4.1). Tests can run on physical devices only locally, while in Cloud they can run only on simulator/ emulator.

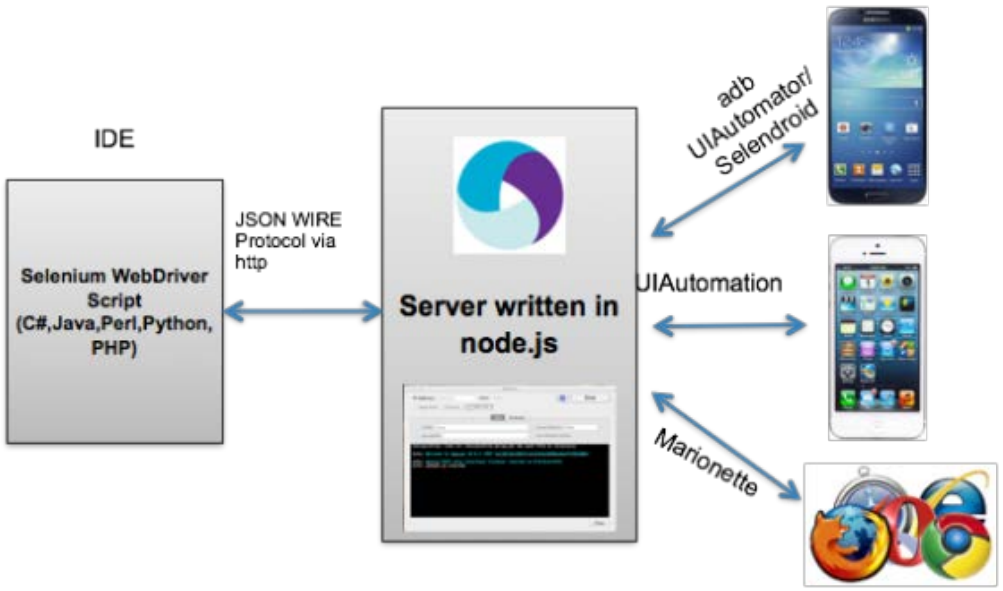


Fig. 4.1. Appium Architecture.[101]

[99] https://msdn.microsoft.com/en-us/library/dd380742.aspx
[100] http://appium.io/
[101] https://domich.wordpress.com/tag/appium/

4.2.2.2. Xamarin Test Cloud

Xamarin Test Cloud supports UI test automation for iOS and Android. Tests for Xamarin Test Cloud can be written using two frameworks[102]:

- Xamarin.UITest – C# tests.
- Calabash – Cucumber ("business language") notation tests.

Test can be executed in Xamarin Test Cloud on the physical devices. In order to achieve this Xamarin Test Cloud agent is installed:

- as a separate app on Android;
- as a library built-in to the app under test on iOS.

"Xamarin Test Cloud Agent should only be included in Debug builds of the application"[102] for iOS. Here it is worth to mention that apps can work differently when they are built in Debug configuration in comparison to the Release configuration. The architecture of Xamarin Test Cloud Agent for the both platforms is shown in Fig. 4.2 and Fig. 4.3.

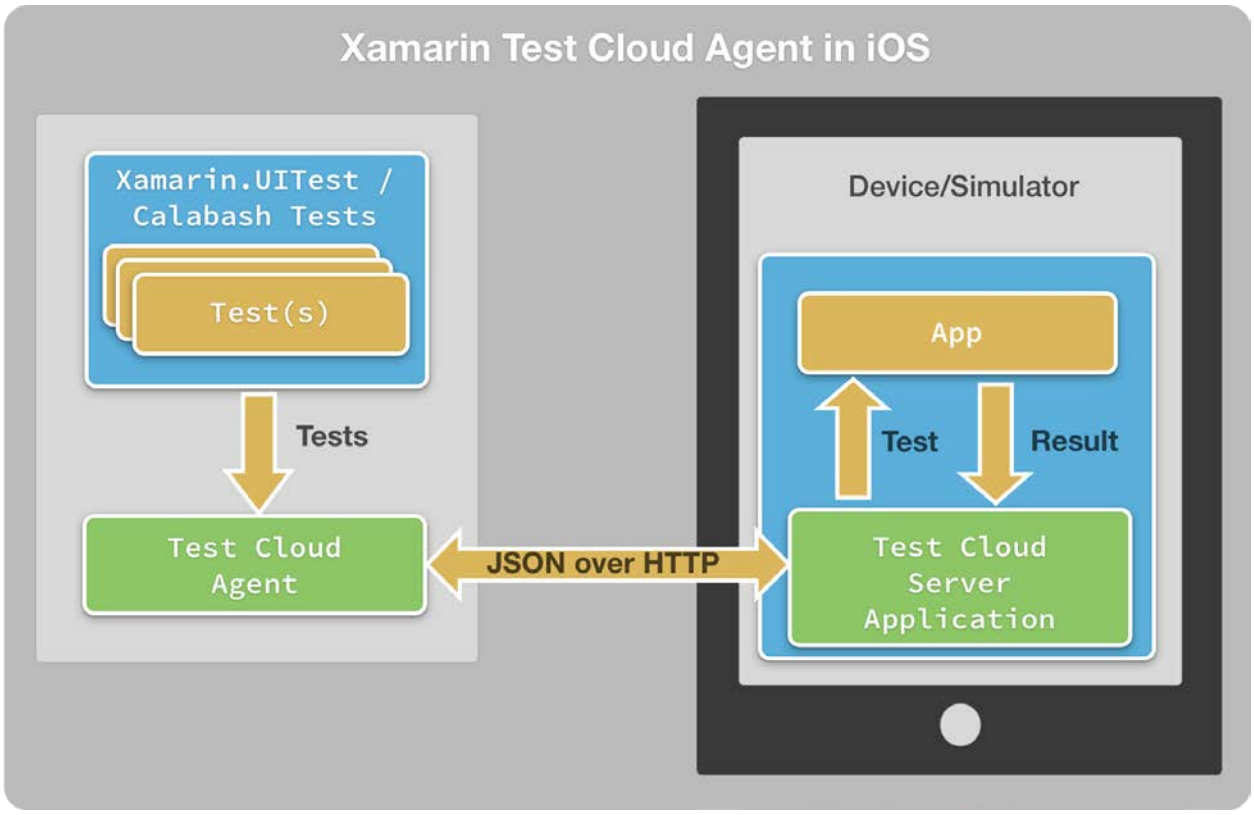


Fig. 4.2. Xamarin Test Cloud Agent in iOS.[102]

[102] https://developer.xamarin.com/guides/testcloud/introduction-to-test-cloud/

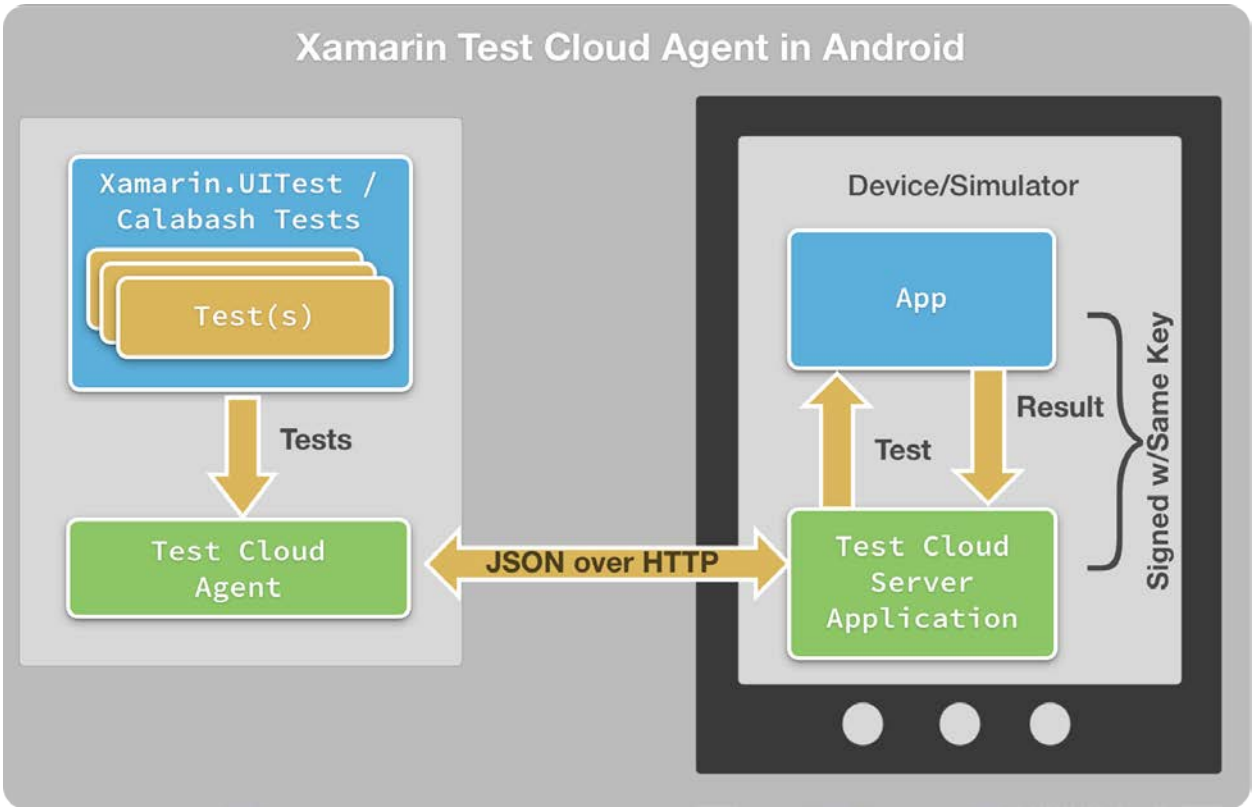


Fig. 4.3. Xamarin Test Cloud Agent in Android.[102]

It is also possible to run Appium tests using Xamarin Test Cloud infrastructure from July, 2015[103].

4.2.2.3. Tosca Mobile+

Tosca Mobile+[104] is on-premises only UI test automation tool for iOS and Android. Tosca Mobile+ is a part of Tricentis Tosca Testsuite test automation tool. Tosca needs adapted MonkeyTalk library (now belongs to Oracle[105] and is not open source anymore) to be integrated into the app for iOS case. This means that app under test and the released app have the different codebase. Tests can be written in VB, C#, VBScript, or through IDE UI. Tosca Mobile+ can also run tests using image recognition framework called Sikuli[106] (see section Sikuli4.2.3.1).

4.2.2.4. Telerik Test Studio Mobile

Telerik Test Studio Mobile[107] is a part of a comprehensive Telerik Test Studio solution. It supports UI test automation for iOS[108] and Android[109]. For both cases custom library to be

[103] https://blog.xamarin.com/xamarin-test-cloud-to-support-appium-framework/
[104] http://www.tricentis.com/tricentis-tosca-testsuite/tosca-mobile-plus/
[105] https://www.oracle.com/corporate/acquisitions/cloudmonkey/index.html
[106] https://www.youtube.com/watch?v=o9n15zkXX24
[107] http://docs.telerik.com/teststudio/test-studio-mobile/overview
[108] http://docs.telerik.com/teststudio/test-studio-mobile/native-applications/configure-your-app/configure-ios
[109] http://docs.telerik.com/teststudio/test-studio-mobile/native-applications/configure-your-app/configure-android

inserted into the app. It also needs an agent app to be installed to the device under test. Furthermore, studio itself runs only on Windows machine[110]. It supports native and web applications testing, while hybrid apps are not supported[111]. Communication with web apps on mobile devices is achieved through Fiddler proxy[112]. Tests are written through Telerik Test Studio GUI or in C# using the plugin for Microsoft Visual Studio[113].

4.2.2.5. DeviceAnywhere

DeviceAnywhere is a test automation solution for iOS and Android apps that allows running the tests on the physical devices in the Cloud. DeviceAnywhere testing lab is depicted in Fig. 4.4. Tests can be written using IDE GUI or using Java API. It is also possible to run Appium tests using their infrastructure. The solution allows taking the video records of the test execution.[114]

Fig. 4.4. DeviceAnywhere Testing Lab.[114]

According to the guide on the vendor's website DeviceAnywhere agent together with some other 3rd party apps to be installed from Cydia store in order to onboard the physical iOS device and take it under the control[115]. It means that tests run on jail-broken device. This fact has many drawbacks:

1) Unpredicted behavior of the app – there is no warranty that app will run in the same way as on a jailed device, but the most users will use the jailed devices.

[110] http://docs.telerik.com/teststudio/test-studio-mobile/overview-mb/native-applications
[111] http://docs.telerik.com/teststudio/test-studio-mobile/native-applications/installation/agent-app-installation
[112] http://docs.telerik.com/teststudio/test-studio-mobile/web-applications/configuration/certificate
[113] http://www.telerik.com/teststudio/visual-studio-testing-plugin-benefits
[114] http://www.keynote.com/solutions/testing/mobile-testing
[115] http://www.keynote.com/go/product-documentation/iOSDeviceOnboardingGuide.pdf

2) It is a requirement in many enterprises to add the check if device is jail-broken. The apps do not start if so. Verification if Cydia app is installed is a part of such check. It means that special app version without such check to be produced for test automation purposes.
3) It always takes a time for hackers to jail-brake the new iOS version. It means that it is not possible to run the tests on the latest iOS version (that is very important in the mobile world) till jail-break is released and till the vendor has updated the agent version to be compatible with the latest OS.

4.2.2.6. Ranorex

Ranorex can automate iOS and Android apps. It "instruments" the apps to make them available for automation. In case of iOS it means that additional code is added to the app binary and app is resigned afterwards. The following note can be found at the framework website: "Because the Ranorex automation lib uses non-public APIs, make sure that you do not submit a Ranorex instrumented app to the app store as your app might be rejected and you might be banned from submitting apps to the app store for a period of time."[116] Another note is added on Android app instrumentation: "…When disabling "Tree simplification", UI-trees will remain unchanged. This means no post processing will take place, resulting in larger UI-trees. Disabling this option decreases the apps startup performance but might be useful when automating 3rd party controls."[117] All this is quite risky because of two facts:

1) The first already known issue that productive app and app under test has different code based.
2) The second issue is that using of non-public APIs and post processing the UI tree can delay the adoption of the test framework for new OS version. It will also need the adoption of the test infrastructure if tests to be run on different OS versions.

Tests can be recorded through IDE or written/ adjusted in C# or VB.Net.[118]

4.2.2.7. SeeTest

SeeTest supports automated testing of iOS, Android, Windows Phone 8.0/ 8.1, and BlackBerry apps. It requires adding the custom library into iOS app under test.[119] It supports not only native/ web object recognition, but also can identify object using image recognition and using OCR technologies to recognize the objects containing the text.

[116] http://www.ranorex.com/support/user-guide-20/instrumentation-wizard/ios.html
[117] http://www.ranorex.com/support/user-guide-20/android-testing.html
[118] http://www.ranorex.com/test-automation-tools.html
[119] https://docs.experitest.com/display/public/SA/Manually+Instrumenting+iOS+Applications

The solution enables the creation of the own onsite test lab that could be accessed through internet. The tests can be written in IDE GUI, C#, Java, Perl, Python, Ruby. The environment is also prepared to run the tests using Appium.

4.2.3. Image Recognition Based Tools

4.2.3.1. Sikuli

Sikuli used to be an open-source project for test automation using image recognition technologies. The tests can be written in Python, Ruby, JavaScript, and Java (Java API is the core of the solution). It is platform independent tool, so it is applicable for mobile automation as well.[120] However, in order to automate the tests on the physical device its screen to be projected on the machine's screen where the tests are run from. It is possible for Android to connect to it through VNC viewer. But there are no official VNC servers available for iOS that can run exactly on the device without jail-breaking it. Right now the project development continues (as SikuliX), but they do not consider to develop the mobile testing anymore.[121]

4.2.3.2. eggPlant

eggPlant is a cross-platform image comparison based test automation tool. iOS Gateway installed on a Mac machine is used as a mobile VNC server to connect eggPlant framework with iOS device. Custom Springboard application (iOS home screen) is provided by eggPlant to get the control on the device.[122] The iOS Gateway network architecture is depicted in Fig. 4.5.

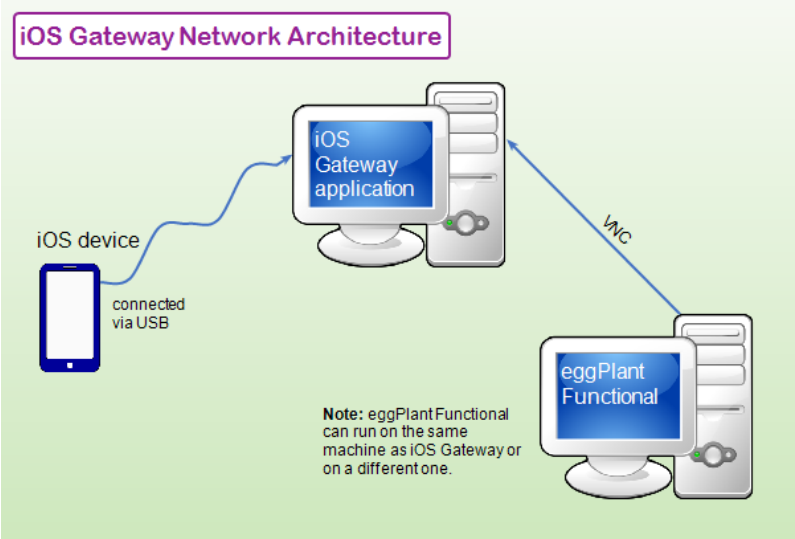


Fig. 4.5. iOS Gateway Network Architecture.[122]

[120] http://www.sikuli.org/testing.html
[121] http://www.sikulix.com/quickstart.html
[122] http://docs.testplant.com/ePF/using/epf-getting-started-ios-gateway.htm

4.2.4. Summary

The market players with characteristics they posses are summarized in Table 4.1., Table 4.2, and Table 4.3. 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.

Table 4.1.

OEM Solutions for Mobile UI Test Automation

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

Table 4.2.

Cross-platform Solutions for Mobile UI Test Automation - Clustering

Name	Wrapper	API-based	Image-based	3rd Party Library in Use (If Not Own)	3rd Party Library Integration into Source Code (for iOS)
Appium	X	X		Implements Selenium WebDriver	
Xamarin Test Cloud		X		Calabash	X
Tosca Mobile+		X		Modified MonkeyTalk, Sikuli	X
Telerik Test Studio Mobile		X			X
DeviceAnywhere		X			X
Ranorex		X			X
SeeTest		X			X
Sikuli			X		
EggPlant			X		

Table 4.3.

Cross-platform Solutions for Mobile UI Test Automation - Characteristics

Name	Device Support	Cloud Support	Scripting Languages	Native/ Hybrid	Web	Costs
Appium	X	+/- (Simulator only)	Java, Ruby, Python, PHP, JavaScript, C#	X	X (comes with wrapper)	Free/ pay for cloud
Xamarin Test Cloud	X	X (can run Appium)	C#, Ruby	X	+/- (need to wrap the website into native app)	Paid
Tosca Mobile+	X	X (private cloud with deviceConnect[123] by MobileLabs)	Through IDE, VB, C#, VBScript	X	X (comes with wrapper)	Paid
Telerik Test Studio Mobile	X	X	Through IDE, C#	+/- (hybrid are not supported)		Paid
Device Anywhere	X	X (can run Appium)	Through IDE, Java	X		Paid
Ranorex	X	X	Through IDE, C#, VB.Net	X		Paid
SeeTest	X	X (can run Appium)	Through IDE, C#, Java, Perl, Python, Ruby	X		Paid
Sikuli	iOS - simulator only	-	Java, Python, Ruby, JavaScript	X	X	Free
EggPlant	X	X	SenseTalk, Java, C#, Ruby	X	X	Paid

[123] http://mobilelabsinc.com/products/deviceconnect/

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

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.

5. TTAP EXTENSION FOR APPLE UIAUTOMATION

5.1. 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. [70]

5.1.1. 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.

5.1.2. 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.

5.1.3. 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.

5.1.4. 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.

5.1.5. 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.

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).

5.1.6. 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 5.1 and Table 5.2.

Table 5.1.

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 5.2.

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

5.2. 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 4.1.
- Took into account the weak points set of each solution described in Chapter 4.
- 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 Xamarin[124] 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.

5.3. The Rise of tTap

5.3.1. Introduction

When doing the first proofs of concepts in UIAutomation we took a look at both of the extensions mentioned in section 4.2.1.1. 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 UIAutomaiton framework (see section 5.1). During the extensive test automation process it appeared that we need the different sets of

[124] http://xamarin.com/

commands in comparison to Tuneup JS to make the test automation process more convenient. That is why we started to cut, rewrite, and extend Tuneup JS extension that resulted into new extension creation that we call tTap[125] – 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 chapter. It is worth mentioning that tTap extension is distributed under MIT license[126].

5.3.2. Solution Details

The goal of tTap is to overcome the limitations of the original UIAutomation aggregated in Table 5.2. 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.

5.3.3. 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. 5.1.). To do this in automatic way tTap contains function that runs AppleScript that “clicks” this menu item using the hotkeys.

[125] https://github.com/ivans-kulesovs/tTap
[126] http://opensource.org/licenses/MIT

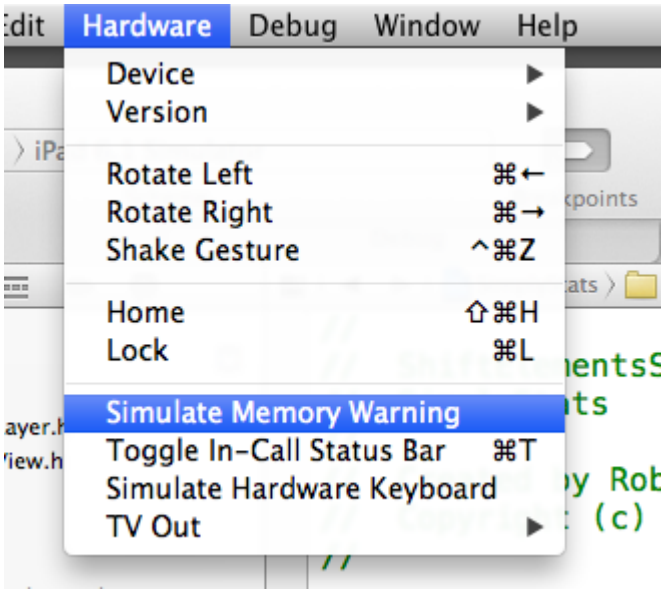


Fig. 5.1. Triggering low-memory warning on the simulator.[127]

5.3.4. 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.

5.3.5. 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. 5.2). 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. 5.3).

[127] http://stackoverflow.com/questions/5323515/unable-to-simulate-invoking-applicationdidreceivememorywarning

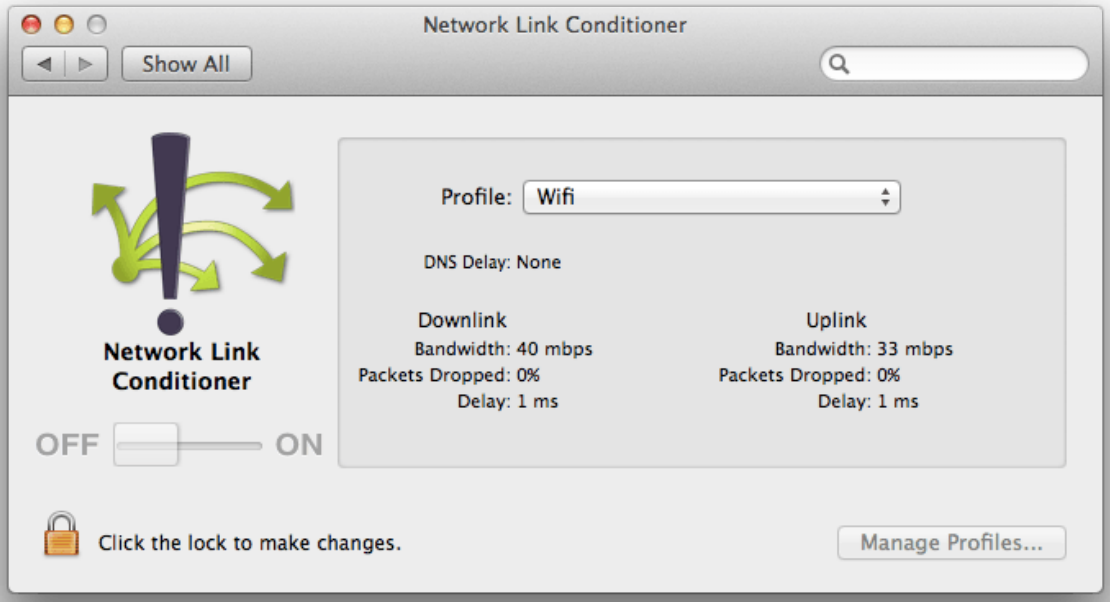


Fig. 5.2. Switching on/ off WiFi connection via LinkConditioner.

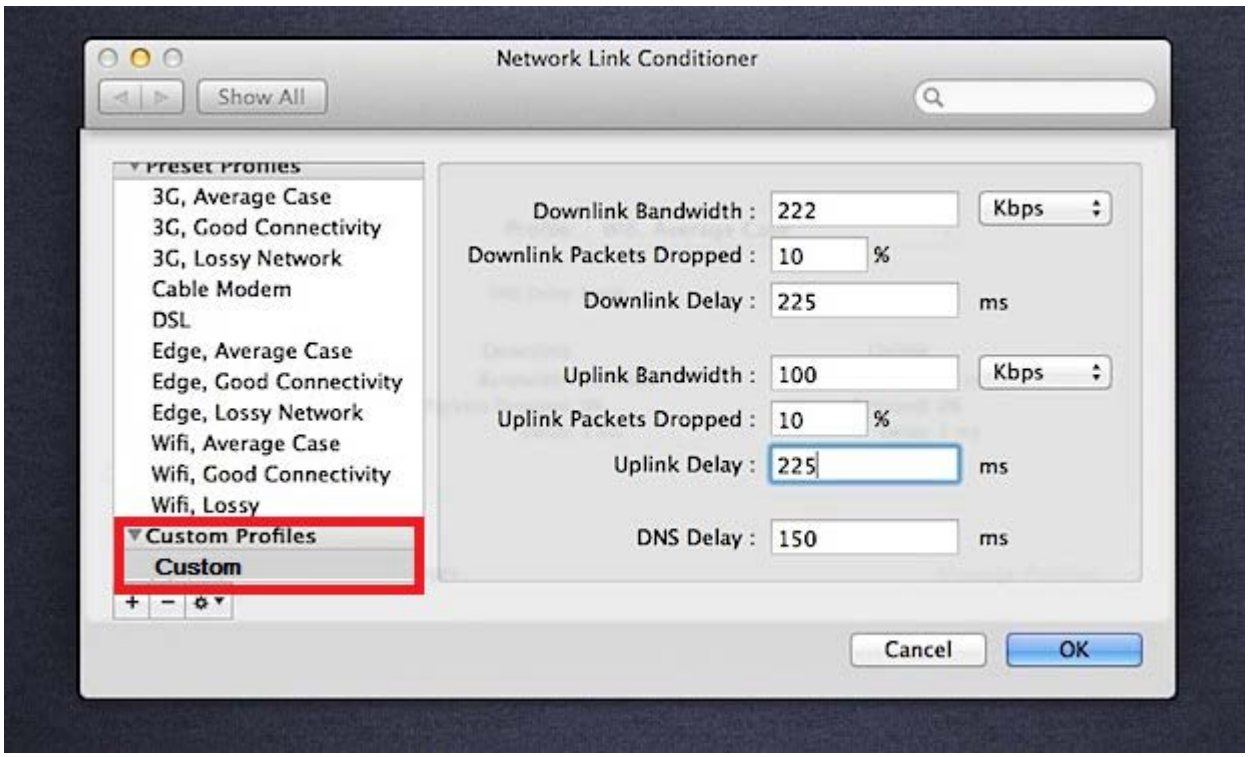


Fig. 5.3. Creating new network conditions profile.

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. 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.

5.3.6. 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 [71].

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.

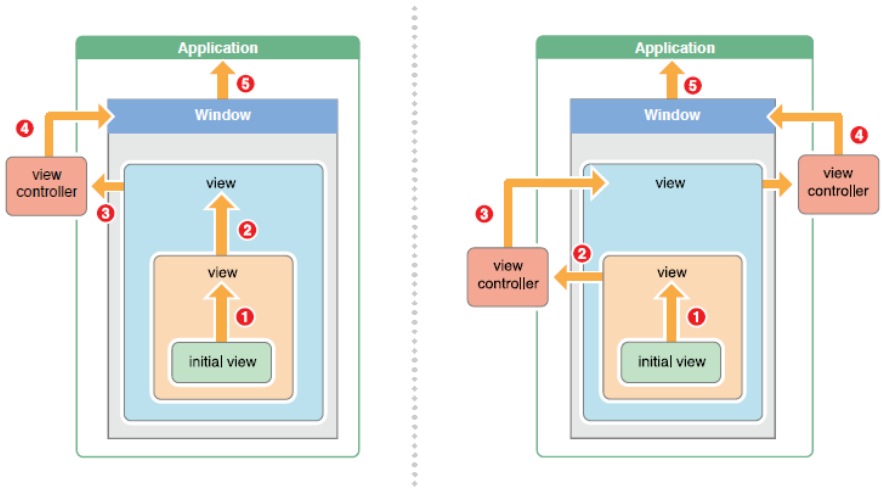


Fig. 5.4. The examples of responder chain in iOS. [72]

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 author and colleagues 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.

5.3.7. 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 5.3*.

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.

Table 5.3

The Status of Overcoming Apple UIAutomation Limitations

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 app	×	✓
Add/ remove contacts from Contacts app	×	✓
Framework		
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	✓	✓

- 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.

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.

5.4. Practical Usage Experience

5.4.1. Connectivity

For many of the mobile apps being online is treated as a default behavior. tTap allows to switch the connectivity to the WiFi shared from the automated tests execution host. Here are the examples of test scenarios that could be automated exclusively using tTap:

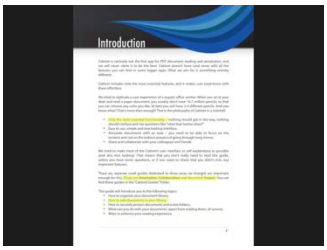
- App startup and usage in offline:
  - check that online functionality is not available;
  - check that invoking online functionality does not break the app;
  - check that appropriate error messages are shown when the online functionality is invoked.
- Losing the connection during the online activity:
  - check that corrupted data is not stored;
  - check that appropriate error message is shown;
  - check that online data can be retrieved after connection is reactivated.

5.4.2. Image Comparison

There are some situations when there is no other chance to test the functionality without using the image comparison, while this solution is thought to be less robust and should not be used without the real need. In our practice we used image comparison in such straight-forward cases:

- When drawing the annotations, i.e. most of the OpenGL activities could be checked like this. The example is depicted in Fig. 5.5.
- When testing the functionality of the special area bookmarks on the large space, i.e. the exact viewport of the definite position and zoom level should be shown when user taps on the bookmark. The examples are depicted in Fig. 5.6 and Fig. 5.7.

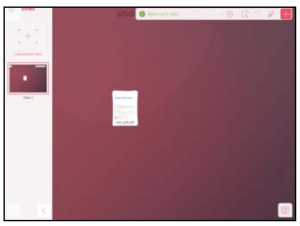
a) Start drawing annotations.



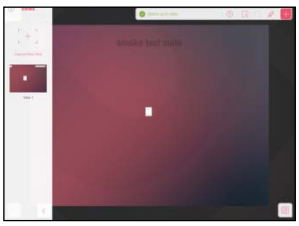
b) Drawn annotations.

c) Comparison result.

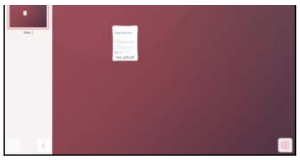
Fig. 5.5. Image comparison example of OpenGL activities.



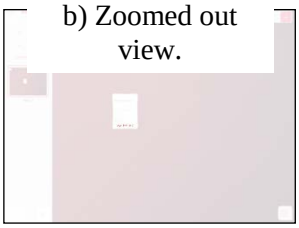
a) Original bookmarked viewport.



b) Zoomed out view.

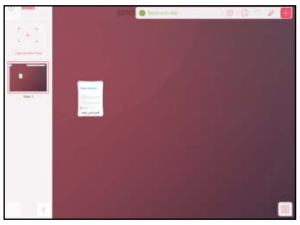


c) View after navigating via bookmark.

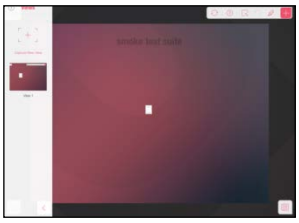


d) Comparison result between 1 and 2. The difference is shown in bright red. Small delta is allowed.

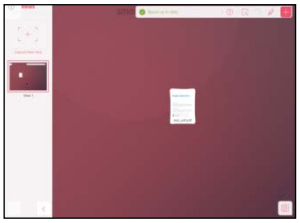
Fig. 5.6. Image comparison passed test example of viewport bookmark functionality.



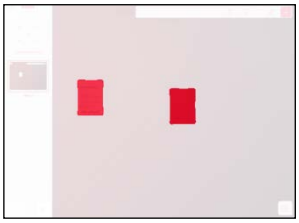
a) Original bookmarked viewport.



b) Zoomed out view.

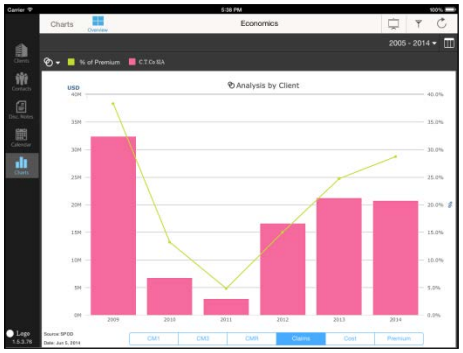


c) View after navigating via bookmark.



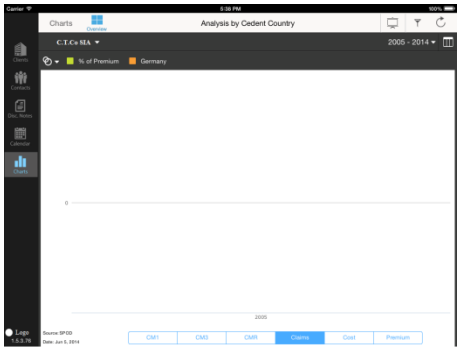
d) Comparison result of 1 and 2. The difference is shown in bright red.

Fig. 5.7. Image comparison failed test example of viewport bookmark functionality.



a) App overview with properly drawn chart.

b) Example when chart is not drawn, but app's logic or HighCharts library has caught the error.



c) Example when chart is not drawn, but neither app's logic, nor HighCharts library has not caught the error. Only comparison of chart area with background does.

Fig. 5.8. Image comparison testing: the example of comparison with background technique.

The image comparison can also be used to check that there is something else on the screen than just the background. For example, we have created the app that draws some financial charts per some client and other filters using the HighCharts[128] JavaScript library. The app is integrated with the server through REST JSON services that convert the data from database to the expected format, while the same data is used for same purpose in some legacy desktop systems. There are already many historical data inside the database. The main goal of the automated testing was to verify that some meaningful charts or the table representation of the same data is shown on the screen. The more we check – the more confidence is in our solution. We did this using two hooks:

- displaying and checking the status that is made visible by the system if app itself or HighCharts library has determined some exception when trying to parse or to display the incoming data;
- comparing the chart or table area with background and logging the warning when the area screenshot is close to background for more than 95% percents; it allowed catching more than 10 defect categories when chart or table was not displayed on the screen while the app logic or chart library did not catch the error.

The example is depicted in Fig. 5.8. The test script was iterating through the different clients, options, filters, etc.

5.4.3. Test Data Setup and Cleanup

Many apps work with images and contacts as data entities. In most cases data is imported from Photos and Contacts app (see Fig. 5.9). To make the tests repeatable the same test data should be available in theses apps during the test setup. This can be achieved only if it is possible to add/ remove images and contacts from Photos and Contacts apps within the automated test execution. tTap provides such capability. Usually all data is being cleaned up from above mentioned apps and then only the test data is added during the test setup. Depending on the other test scenarios related to the same functionality test data could be also cleaned up in each test tear down.

[128] http://www.highcharts.com/

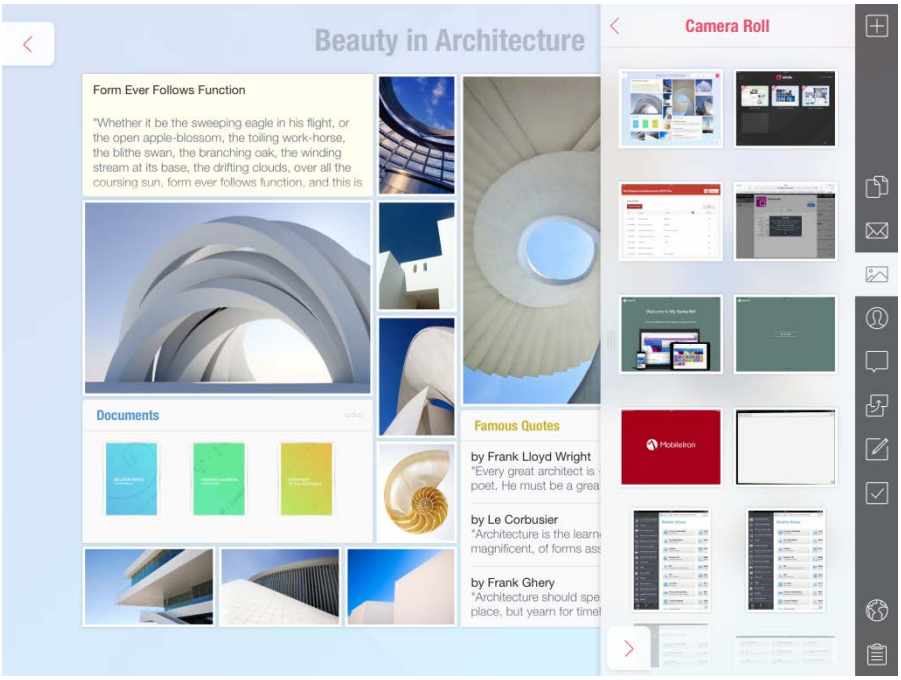


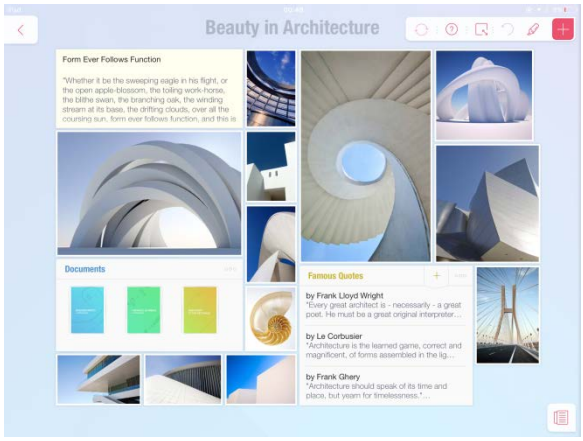
Fig. 5.9. Import from Photos app example.

5.4.4. Framework

Usage of wait conditions and searching for UI element within the whole UI elements tree are framework level features worth demonstrating. There are two most needed wait conditions:

- wait until element is visible – usually is used when screen change happens (see Fig. 5.10);

a) Space selection screen.

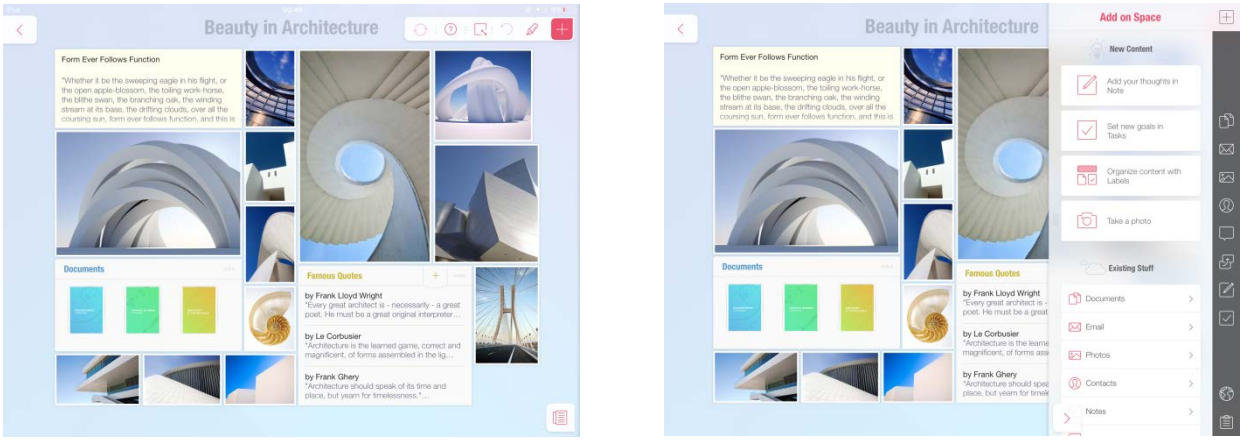


b) Selected space is opened. Wait until the specific space element is visible condition is used to increase autotests robustness.

Fig. 5.10. Wait until element is visible example.

- wait until element has reached the specific position – usually is used when opening some menu, because it can happen that menu is already initialized, but its position

is outside of the screen and it appears in the viewport with some delay due animation (see Fig. 5.11).



a) Side menu is closed.

b) Side menu is opened. Wait until the menu element reaches the specific position condition is used to increase autotests robustness.

Fig. 5.11. Wait until element has reached the specific position example.

In the original UIAutomation we would need to provide the whole path till side panel buttons (for example till Back button of side menu in Fig. 5.11) during test design. This path is not very stable due to hierarchy can change during the future development.

With tTap we can search for the side panel element and then to search the button element inside:

```
getContentBrowser: function(){
      var contentBrowser = searchByName(SPACE_CONTENT_BROWSER);
      if (contentBrowser instanceof UIAElementNil) return null;
      else return contentBrowser;
};

getBackButton: function(){
      return searchByName(SPACE_CONTENT_BROWSER_BUTTON_BACK, this.getContentBrowser());
};

function searchByName(name, startElement){
      var predicate = predicateWithFormat("name = %@", name);
      return searchWithPredicate(predicate, startElement);
};
```

It will always found the right element irrespectively of the UI changes (unless side menu and buttons inside it are still present). Definition of function `searchWithPredicate(predicate, startElement, fDelay)` can be found in Appendix D.

5.4.5. Device vs. Simulator

During the test framework adaption for the real enterprise needs we faced some issues that resulted into decision to run daily or nightly tests only on the real device. The simulator should be used only for test design. First of all, there are a couple of things that just does not work on

simulator, e.g. pinch to zoom inside the scroll view. This was broken starting from iOS7. We have reported this to Apple, but they said that our bug is a duplicate. Now it is iOS 9, but pinch to zoom inside the scroll view still does not work on a simulator. There is also a case when buttons on the system modal windows, e.g. native email controller, responded to the automation on simulator only after they were tapped manually for the first time. Of course, such issues are not acceptable for the continuous integration (CI). The usage of Xamarin cross-platform solution could be the reason for unresponsive buttons, but the same works properly on a device.

Another issue with running tests on simulator in CI environment is that execution time and simulator responsiveness depends on the load on the CI machine. Running tests on CI machine under load on simulator will result into the unrepeatable test failures for sure.

The app under test can crash on the device easier than on the simulator due to the memory management things. It is good when memory leaks are found during the long test runs. However, app can also crash or stall during the test execution because of the internal networking queue on the device (test commands are sent to the device through the internal Bonjour networking server). Execution of some gestures by UIAutomation (e.g. swipe gesture) is more memory consuming then the manual one (we have performed the experiment where swiping of photos by UIAutomation crashed the app after 10-15 swipes, while the same could not be achieved during the manual test run). That is why it is a good idea to split tests into smaller (e.g. 5 minutes) test suites, if possible, and to restart the app between two different test suites execution. This helps to decrease the number of unrepeatable test failures.

It is worth mentioning, that UIAutomation speed decreases during the long test runs and it can fail unexpectedly at the end (when collecting and saving the test run data) when they are executed through the UI of the tool, while we have not experienced such unexpected failures when running the tests from the command line. However, as mentioned earlier, splitting the test to the smaller test suites can help to fix this issue as well.

The great difference between device and simulator is that device has ARM processor, while simulator runs on machine with x86 processor. For example, displaying the formatted HTML text and using of OpenGL on simulator works in the freezing manner, while the same works fine on the device. Another example could be the difference in precision of epsilon value on different architectures. Epsilon is the smallest positive float value. [73], [74] There are much more differences when running app in the environments with different architecture. That is why the results of the tests can just be different in some particular case, but we focus, of course, on the apps to work properly on the real device.

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

The analysis of the mobile UI test automation tools from the fourth chapter, 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 Winium[129] started that wraps Coded UI Tests as well. The proposal is schematically depicted in Fig. 5.9.

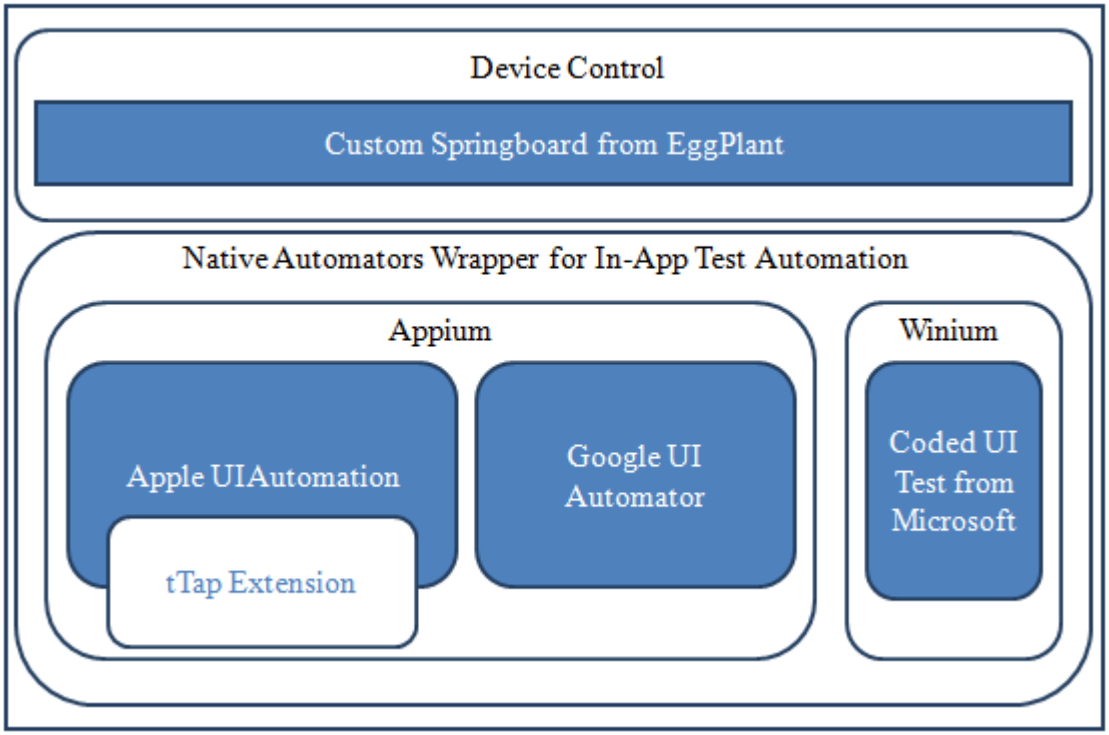


Fig. 5.12. The ideal cross-platform mobile UI test automation tool architecture.

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.

[129] https://github.com/2gis/Winium.StoreApps.CodedUi

CONCLUSIONS AND DISCUSSIONS

The thesis consists of five 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 vision, different (organizational and project-wide) contexts, testing approach, testing method, testing technique, testing plan, etc. Uniting various software testing processes under software testing tactic term has been made for the first time. The visualization of the relation and clustering 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 five 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 details of each aspect were discussed based on the selected sources and the author's professional experience giving the appropriate references to Apple Developers Library3 and other credible sources. Some aspects that were not identified through literature reviews, but are known to the author (iAd, update of Xcode, AirDrop, etc.) were discussed as well.

The author concludes that the study eliminates the gap that existed in the academic world in regards to the identification and detailed description of iOS application testing aspects. Some of the information on a single aspect is available, however there is no comprehensive

systematization with explanation performed before, especially using the systematic literature review as a research method. These details should also be useful for practitioners who want to make their iOS testing strategy more solid and complete.

The third part looks into the functional security testing of mobile 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 author 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. It is worth mentioning that term wrapper (for Appium like architecture) is introduced by the author. There is also no discussion has been held before why integrating the 3rd party library to the app code to enable automation is not a way to go.

The rigorous study of the capabilities and limitation of Apple UI Automation has been done for the first time. This is reflected in the fifth chapter. 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 overcoming of these limitations enabled tTap to be used for iOS apps test automation in C.T.Co software development company.

The practical usage experience of the framework is summarized in the fifth chapter as well. The examples of conditions where image comparison is the only or the most efficient way for assertion are given. The examples of the most used wait conditions and test scenarios where change of connectivity is needed are provided as well. Other examples including the test setup/ teardown with such data entities as images and contacts from Photos and Contacts app are also described. The benefit of improvement that allows searching for UI element within the whole UI elements tree is shown. The arguments why testing should be performed mostly on the real devices are summarized – most of them are related to the difference between ARM (real device) and x86 (simulator) processors architecture and to the difference in the memory management and consumption.

The investigations, analysis, and tTap solution creation from the fourth and fifth chapters 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.

To conclude, the author makes a step forward in software testing terminology through providing the clear definition of such terms as testing approach, testing method, and testing techniques using the solution made by Anthony in the field of language teaching. 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 author investigations on the mobile testing aspects, mobile 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.

REFERENCES

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

[2] J. Barzdin, J. Bicevskis and A. Kalninsh, "Construction of Complete Sample Systems for Program Testing," *Ucenye Zapiski Latv. Gos. Univ.*, vol. 210, pp. 152-188, 1974.

[3] J. Barzdin, J. Bicevskis and A. Kalninsh, "Automatic Construction of Complete Sample System for Program Testing," in *IFIP Congress*, 1977.

[4] G. Myers, The Art of Software Testing, 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1979/ 2004.

[5] B. Beizer, Software Testing Techniques, 2nd ed., New York: Van Nostrand Reinhold Co., 1990.

[6] B. Beizer, Black-Box Testing: Techniques for Functional Testing of Software and Systems, New York: John Wiley & Sons, Inc., 1995.

[7] C. Kaner, J. Falck and H. Nguyen, Testing Computer Software, 2nd ed., John Wiley & Sons, Inc., 1999.

[8] C. Kaner, J. Basch and B. Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach, New York: John Wiley & Sons, Inc., 2001.

[9] B. Pettichord, "Schools of Software Testing," 2007. [Online]. Available: http://www.prismnet.com/~wazmo/papers/four_schools.pdf.

[10] R. Black, Advanced Software Testing – Vol.1, Santa Barbara, CA: Rock Nook Inc., 2009.

[11] Research and Markets, "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)," 2014. [Online]. Available: http://www.researchandmarkets.com/research/7pj4cv/mobile_cloud. [Accessed 21 February 2016].

[12] Markets and Markets, "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," 2014. [Online]. Available: http://www.marketsandmarkets.com/Market-Reports/heterogeneous-mobile-processing-

computing-market-173926586.html. [Accessed 21 February 2016].

[13] Citrix, "Citrix Data Reveal New Global Trends in Consumer and Enterprise Mobility," 2015. [Online]. Available: http://www.citrix.com/news/announcements/feb-2015/citrix-data-reveal-new-global-trends-in-consumer-and-enterprise-.html. [Accessed 21 February 2016].

[14] Good Technology, "Good Technology™ Mobility Index Report Q4 2013," 2013. [Online]. Available: https://media.good.com/documents/rpt-mobility-index-q413.pdf. [Accessed 21 February 2016].

[15] Crittercism, "Mobile Experience Benchmark," 2014. [Online]. Available: http://pages.crittercism.com/rs/crittercism/images/crittercism-mobile-benchmarks.pdf. [Accessed 21 February 2016].

[16] T. Eston, "Android vs. Apple iOS Security Showdown," 2012. [Online]. Available: http://pittsburgh.issa.org/Archives/Android-vs-iOS-MayUpdate.pdf. [Accessed 21 February 2016].

[17] A. Scott, "Introducing the software testing ice-cream cone (anti-pattern)," 2012. [Online]. Available: http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/. [Accessed 21 February 2016].

[18] I. Kuļešovs, V. Arnicane, G. Arnicans and J. Borzovs, "Inventory of Testing Ideas and Structuring of Testing Terms," *Baltic J. Modern Computing*, vol. 1, no. 3-4, pp. 210-227, 2013.

[19] A. Kalninsh and J. Borzovs, Inventarizacija idej testirovanija programm, Riga, Latvia: Latv. Gos. Univ., 1981.

[20] ISTQB, "Standard glossary of terms used in Software Testing," 2012.

[21] G. Arnicans, D. Romans and U. Straujums, "Semi-automatic Generation of a Software Testing Lightweight Ontology from a Glossary Based on the ONTO6 Methodology," in *Frontiers in Artificial Intelligence and Applications. Databases and Information Systems VII: Selected Papers from the Tenth International Baltic Conference*, vol. 249, IOS Press, 2013, pp. 263-276.

[22] G. Arnicans and U. Straujums, "Transformation of the Software Testing Glossary into a browsable Concept Map," in *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)*, 2012.

[23] D. Hoffman, "A Taxonomy for Test Oracles," *Quality Week*, 1998.

[24] ISO, *ISO/IEC 25010:2011*, 2011.

[25] P. Amman and J. Offutt, Introduction to Software Testing, Cambridge: Cambridge University Pres., 2008.

[26] R. Pressman, Software Engineering: A Practitioner's Approach, 6th ed., Singapore: McGraw-Hill.

[27] J. Sommerville, Software Engineering, 8th ed., Harlow, Essex: Pearson Education Limited, 2007.

[28] B. Kumaravadivelu, Understanding Language Teaching: From Method to Postmethod, Mahwah, NJ: Lawrence Erlbaum Associates, Inc., 2006.

[29] G. Hall, Exploring English Language Teaching: Language in Action., New York: Routledge, 2011.

[30] L. Copeland, A Practitioner's Guide to Software Test Design, Norwood, MA: Artech House, Inc., 2003.

[31] K. Tatsumi, "Test Case Design Support System," in *Proceedings of International Conference on Quality Control (ICQC)*, Tokyo, 1987.

[32] R. Hamlet, "Random Testing," in *Encyclopedia of Software Engineering*, Chichester, Wiley, 1994, pp. 970-978.

[33] P. Jorgensen, Software Testing: A Craftsman’s Approach, 3rd ed., Boca Raton, FL: Auerbach Publications, 2008.

[34] Z. Bicevska, J. Bicevskis and I. Oditis, "Smart technologies for improved software maintenance," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, Lodz, Poland, 2015.

[35] J. Bicevskis, Z. Bicevska and I. Oditis, "Self-management of information systems," in *12th International Baltic Conference on Databases and Information Systems*, Riga, 2016.

[36] K. Rauhvarger and J. Bicevskis, "Towards a Semantic Execution Environment Testing," *Scientific Papers, University of Latvia*, vol. 733, pp. 38-52, 2008.

[37] E. Diebelis and J. Bicevskis, "Software Self-Testing," *Frontiers in Artificial Intelligence and Applications*, vol. 249: Databases and Information Systems VII, pp. 249-262, 2013.

[38] I. Oditis and J. Bicevskis, "Asynchronous Runtime Verification of Business Processes: Proof of Concept," *International Journal of Simulation - Systems, Science & Technology*, vol. 16, no. 6, pp. 1-11, 2015.

[39] I. Oditis and J. Bicevskis, "Asynchronous Runtime Verification of Business Processes," in *Proceedings - 7th International Conference on Computational Intelligence, Communication Systems and Networks*, Riga, 2015.

[40] H. Muccini, F. Di Antonio and P. Esposito, "Software testing of mobile applications: challenges and future research directions," in *Proc. IEEE 7th Int. Workshop Automation of Softw. Test*, Zurich, 2012.

[41] V. Dantas, F. Marinho, A. da Costa and R. Andrade, "Testing requirements for mobile applications," in *Proc. 24th Int. Symp. Comput. and Inform. Sci.*, Guzelyurt, 2009.

[42] J. Gao, B. Xiaoying, T. Wei-Tek and T. Uehara, "Mobile application testing: a tutorial," *IEEE Computer,* vol. 47, no. 2, pp. 46-55, 2014.

[43] D. Franke, C. Elsemann, S. Kowalewski and C. Weise, "Reverse engineering of mobile application lifecycles," in *Proc. 18th Work. Conf. Reverse Eng.*, Limerick, 2011.

[44] D. Franke, S. Kowalewski, C. Weise and N. Prakobkosol, "Testing conformance of lifecycle-dependent properties of mobile applications," in *Proc. 5th Int. Conf. Softw. Testing, Verification and Validation*, Montreal, 2012.

[45] R. Ogawa and B. Malen, "Towards rigor in reviews of multivocal literatures: applying the exploratory case study method," *Review of Educ. Research,* vol. 61, no. 3, p. 265–286, 1991.

[46] I. Kulesovs, "iOS Applications Testing," in *Environment. Technology. Resources. Proceedings of the 10th International Scientific and Practical Conference.*, Rezekne, Latvia, 2015.

[47] E. Tom, A. Aurum and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software,* vol. 86, no. 6, pp. 1498-1516, 2013.

[48] B. Kitchenham and S. Charters, Guidelines for performing Systematic Literature Reviews in Software Engineering, EBSE Tech. Rep., 2007.

[49] D. Franke, S. Kowalewski and C. Weise, "A mobile software quality model," in *Proc. 12th Int. Conf. Quality Softw.*, Xi'an, Shaanxi, 2012.

[50] H.-K. Kim, "Mobile applications software testing methodology," *Commun. in Comput. and Inform. Sci.,* vol. 342, pp. 158-166, 2012.

[51] E. H. Marinho and R. Resende, "Quality factors in development best practices for mobile applications," in *Proc. Computational Sci. and Its App.,* Salvador de Bahia, Brazil, 2012.

[52] D. Amalfitano, A. Fasolino, P. Tramontana and N. Amatucci, "Considering context events

in event-based testing of mobile applications," in *Proc. IEEE 6th Int. Conf. Softw. Testing, Verification and Validation Workshops*, Luxembourg, 2013.

[53] K. Haller, "Mobile testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 38, no. 6, pp. 1-8, 2013.

[54] H. Khalid, "On identifying user complaints of iOS apps," in *Proc. 35th Int. Conf. Softw. Eng.*, San Francisco, CA, 2013.

[55] D. Franke and C. Weise, "Providing a software quality framework for testing of mobile applications," in *Proc. 4th Int. Conf. Softw. Testing, Verification and Validation*, Berlin, 2011.

[56] App Quality Alliance, "Testing Criteria for iOS Apps," 22 October 2013. [Online]. Available: http://www.appqualityalliance.org/files/AQuA_testing_criteria_for_iOS_for_v1.0%20final%2022_oct_2013.pdf. [Accessed 23 February 2016].

[57] P. Pound, "Tips For Accessibility Testing Of iOS Apps," 24 May 2013. [Online]. Available: http://patstapestry.wordpress.com/2013/05/24/tips-for-accessibility-testing-of-ios-apps/. [Accessed 23 February 2016].

[58] Nearsoft, "Testing iOS Apps for Tough Network Conditions," 20 October 2013. [Online]. Available: http://nearsoft.com/blog/testing-ios-apps-for-tough-network-conditions/. [Accessed 23 February 2016].

[59] TestElf, "We Find These Common Bugs When Testing iOS Apps," 24 July 2013. [Online]. Available: http://blog.testelf.com/post/56341438836/we-find-these-common-bugs-when-testing-ios-apps. [Accessed 23 February 2016].

[60] uTest, "The Essential Guide to iPhone & iPad App Testing," October 2013. [Online]. Available: http://qawiki.devsmm.com/wp-content/uploads/2014/10/uTest_Whitepaper_The_Essential_Guide_to_iOS_App_Testing.pdf. [Accessed 23 February 2016].

[61] Neglected Potential, "iOS Testing mind map 1.2 – Now with more stuff," 8 October 2013. [Online]. Available: http://www.neglectedpotential.com/2013/10/ios-testing-mind-map-1-2/. [Accessed 23 February 2016].

[62] D. Addey, "iOS Devices," 22 September 2013. [Online]. Available: http://daveaddey.com/postfiles/AgantReleaseChecklist2013.pdf. [Accessed 23 February 2016].

[63] R. Land, "iOS Accessibility - A Useful Guide For Testing," 2 September 2013. [Online].

Available: http://www.rosiesherry.com/2012/09/02/ios-accessibility-a-useful-guide-for-testing/. [Accessed 23 February 2016].

[64] SmartBear, "Testing iOS Applications," 12 March 2014. [Online]. Available: http://blog.smartbear.com/mobile/testing-ios-applications/. [Accessed 23 February 2016].

[65] OpenSignal, "Android Fragmentation Visualized," 2015. [Online]. Available: http://opensignal.com/reports/2015/08/android-fragmentation/. [Accessed 26 March 2016].

[66] I. Kulesovs, J. Borzovs, A. Susters, V. Arnicane, G. Arnicans, K. Keiduns and J. Skutelis, "The Multi-Edge Approach for iOS Applications Testing," in *New Perspectives from User Interfaces and Semantic Web: Information Quality, Advanced Interdisciplinary Applications and Combination of the Technologies Challenges*, Bergamo, Italy, Blue Herons, 2015, pp. 77-107.

[67] I. Gorbans, I. Kulesovs, J. Buls and U. Straujums, "The Myths about and Solutions for an Android OS Controlled and Secure Environment," in *Environment. Technology. Resources. Proceedings of the 10th International Scientific and Practical Conference.*, Rezekne, 2015.

[68] J. Buls, I. Gorbans, I. Kulesovs and U. Straujums, "The Adaptation of Shamir's Approach for Increasing the Security of a Mobile Environment," *Baltic Journal of Modern Computing,* vol. 4, no. 1, pp. 51-58, 2016.

[69] Apple Developer, "UI Automation JavaScript Reference for iOS," 19 September 2012. [Online]. Available: https://developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAutomationRef/. [Accessed 25 February 2016].

[70] I. Kulesovs, A. Susters, K. Keiduns and J. Skutelis, "Automated Testing of iOS Apps: tTap Extension for Apple UIAutomation," in *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, 2015.

[71] J. Penn, Test iOS Apps with UI Automation: Bug Hunting Made Easy, Dallas, Texas: The Pragmatic Bookshelf, 2013.

[72] Apple Developer, "Event Handling Guide for iOS," 2 February 2015. [Online]. Available: https://www.hitpages.com/doc/6281807121088512/1. [Accessed 25 February 2016].

[73] MSDN, "Single.Epsilon Field," [Online]. Available: https://msdn.microsoft.com/en-us/library/system.single.epsilon(v=vs.110).aspx. [Accessed 25 February 2016].

[74] MSDN, "Double.Epsilon Field," [Online]. Available: https://msdn.microsoft.com/en-us/library/system.double.epsilon(v=vs.110).aspx. [Accessed 25 February 2016].

[75] L. Zhifang, "Test Automation on Mobile Device," in *Proc. Proceedings of the 5th Workshop on Automation of Software Test*, 2010.

APPENDIX A: THE FULL LIST OF MULTIVOCAL LITERATURE

The full list of the reviewed multivocal sources with indicating the exclusion phase can be found here: *https://dspace.lu.lv/dspace/bitstream/handle/7/2739/iOS Applications Testing - Multivocal Sources.pdf*

APPENDIX B: TTAP SOURCE CODE

tTap source code can be found here: *https://github.com/ivans-kulesovs/tTap*.

APPENDIX C: LINKCONDITIONER USAGE APPLESCRIPT

```
--> Srart conditioner app
on activateConditioner()
           tell application "System Preferences"
             activate
             set the current pane to pane id "com.apple.Network-Link-Conditioner"
           end tell
end activateConditioner

--> enable for UIAutomation
on activatePreferencesForAutomation()
           tell application "System Events" to tell process "System Preferences"
             set frontmost to true
             delay 1
           end tell
end activatePreferencesForAutomation

on activateConditionerForAutomation()
           activateConditioner()
           activatePreferencesForAutomation()
end activateConditionerForAutomation

-->change the preset
on selectPreset(presetName)
           tell application "System Events" to tell process "System Preferences"
             tell window "Network Link Conditioner" to tell group 1 to tell pop up
button 1
               click
               delay 1
               tell menu 1
                 menu item presetName click
               end tell
             end tell
           end tell
end selectPreset

-->switch ON
on switchOn()
           tell application "System Events" to tell process "System Preferences"
             tell window "Network Link Conditioner"
               button "ON" click
             end tell
           end tell
```

```
end switchOn

-->Switch OFF
on switchOff()
           tell application "System Events" to tell process "System Preferences"
             tell window "Network Link Conditioner"
               button "OFF" click
             end tell
           end tell
end switchOff
```

APPENDIX D: SEARCH WITH PREDICATE FUNCTION

```
function searchWithPredicate(predicate, startElement, fDelay) {
    if (!fDelay) fDelay = 1;
    var target = UIATarget.localTarget();
    var timeoutInMillis = fDelay * 1000;
    var start = new Date();

    function recursiveSearch(predicate, startElement) {
        target.pushTimeout(0);
        var elements = startElement.elements();
        var found = elements.firstWithPredicate(predicate);
        target.popTimeout();

        if (found.isValid()) return found;

        for (var i = 0; i < elements.length; i++) {
            var element = elements[i];
            found = recursiveSearch(predicate, element);
            if (found) return found;
        }
        return null;
    };

    if (typeof startElement == "undefined") {
        //log("search predicate: " + predicate + ", startElement undefined - use
default!");
        startElement = target.frontMostApp().mainWindow();
    } else if (startElement == null) {
        //log("search predicate: " + predicate + ", startElement null - won't
search!");
        return null;
    } else {
        //log("search predicate: " + predicate + ", startElement: " +
startElement.name());
    }

    do {
        var now = new Date();
        var found = recursiveSearch(predicate, startElement);
        target.delay(0.1);
    } while(!found && now - start < timeoutInMillis);

    return found;
};
```