Automated Testing of iOS Apps: tTap Extension for Apple UIAutomation

Ivans Kulesovs
Faculty of Computing,
University of Latvia
19 Raina Blvd.,
Riga, LV-1586, Latvia
Enterprise 2.0,
SIA "C.T.Co"
15/25 Jurkalnes St.
Riga, LV-1046, Latvia
+371 27150595
ivans.kulesovs@gmail.com

Aigars Susters
Enterprise 2.0,
SIA "C.T.Co"
15/25 Jurkalnes St.
Riga, LV-1046, Latvia
+371 26118622
aigars.susters@gmail.com

ABSTRACT
Mobile apps tend to extend or even substitute the existing IT solutions. In the corporate world, according to the statistics, the preference is given to iOS devices. The complexity of the apps increases together with the apps quantity. This also increases the need for automated testing. In a course of the study we compare the existing test automation solutions for iOS apps and describe some extensions for Apple UIAutomation tool. We have also created our own extension called tTap that improves the existing ones and solves several issues that does not work in the clean UIAutomation. We describe the implementation details of our extension and share the practical experience of testing iOS apps using it. We also describe the part of our test lab solution, while the description of the full test lab is planned in the consecutive studies.

Categories and Subject Descriptors
I.2.2 Automatic Programming: Program verification.

D.2.4 Software/Program Verification: Validation, Reliability.

D.2.5 Testing and Debugging: Testing tools.

General Terms
Verification, Reliability.

Keywords
iOS, test automation, mobile, apps.

1. INTRODUCTION
Testing is one of the important parts of the software development process. 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 following test coverage model 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. [6] The model is depicted in Figure 1.

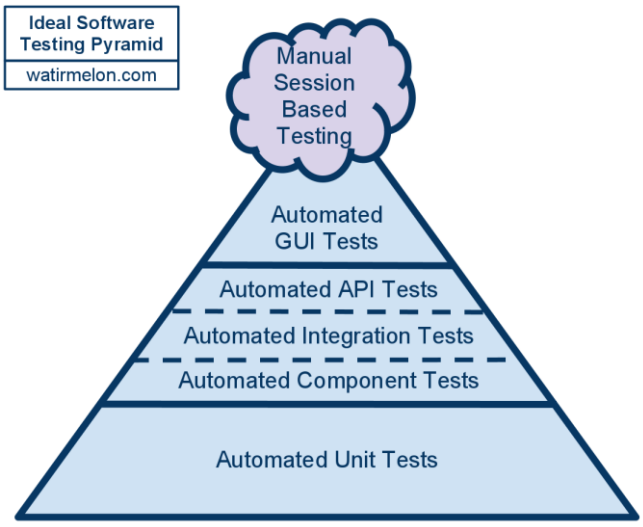


Figure 1. Automated test coverage model per test level [6].

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.

iOS from Apple is one of the most popular mobile operating systems. According to [1], iOS holds 64%, but according to [4], iOS holds even 73% market share of the enterprise mobile devices. According to the same [4], iPads hold 91,4% of enterprise tablets. The authors also work for the company that produces the mobile apps, mostly iOS native, both for enterprise and public ones. The enterprise apps are made for the several Fortune 500 companies.[1] These are the reasons why the UI test automation for iOS native apps was chosen as the main topic of the study.

Conference'10, Month 1–2, 2010, City, State, Country.

[1] - http://money.cnn.com/magazines/fortune/global500/index.html

In a course of the study the extension to Apple UIAutomation[2] framework called tTap [3] was created. The study consists from the 3 more sections. The Section II is the background study on the different UI test automation options available for the iOS apps. The tTap extension for Apple UIAutomation and its usage patterns are described in the Section III. The Section IV concludes the study and sets the goals for the future.

2. BACKGROUND STUDY

2.1 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 OS manufacturer IDE, i.e. UIAutomation by Apple for iOS or uiautomator[4] by Google for Android. All other mobile automation tools are the 3rd party solutions. These third party 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. 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. 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. The main market players with characteristics they posses are shown in Table 1.

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

Apple OEM automation tool is the most robust one between the API-based tools. It comes with a sufficient set of functions to build the commonly used test patterns, but the scripting is too wordy. It is also limited to the one platform.

Wrappers are cross-platform solutions, some of them even come with cloud testing support. But they 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.

We have considered the following when selecting the proper tool for UI test automation for iOS apps developed within our company:

- 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[5] 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.

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.

Before choosing UIAutomation as the tool to automate UI tests with, we have done the following:

- Investigated each solution from Table 1
- Took into account the weak points set of each solution described above
- Took into account the particular environmental options within our company

When choosing the right tool we have acknowledged the limited debugging capabilities of UIAutomation due to the own, non standard JavaScript environment where tests are executed.

2.2 Existing Extensions for Apple UIAutomation

UIAutomation tests are written in JavaScript. The framework consists of the most basic functions for all UI elements available in iOS. [8] 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. The problem lies in creating the commonly used test notations from these basic functions that are also quite wordy. It means that the goal of each extension is to create an 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.

---

[2] - https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html

[3] - https://github.com/ivans-kulesovs/tTap

[4] - http://developer.android.com/tools/testing/testing_ui.html

[5] - http://xamarin.com/

Table 1. The main market players for iOS UI test automation

Solution Name\ Characteristic	**UIAutomation**	**Appium**[6]	**Xamarin TestCloud**[7]	**Tosca Mobile**[8]	**Cloud Monkey**[9]	**Sikuli**[10]	**eggPlant**[11]
OEM	X						
Wrapper		X					
3rd party library in use		Implements Selenium WebDriver	Calabash	Modified MonkeyTalk, Sikuli	MonkeyTalk		
Cross-platform		X	X	X	X	X	X
API-based	X	X	X	X	X		
Image-based				X		X	X
Needs 3rd party library in source code			X	X	X		
Costs	Comes together with Xcode	Free/ pay for cloud	Paid	Paid	Free/ pay for cloud	Free	Paid
Device support	X	X	X	X	X		X
Simulator support	X	X	X	X	X	X	X
Cloud support		X (Simulator only)	X	X (private cloud with deviceConnect[12] by MobileLabs)	X		X
Script languages	JavaScript	Java, Ruby, Python, PHP, JavaScript, C#	C#, Ruby	Through IDE, VB, C#, VBScript	Java, JavaScript, MonkeyTalk commands	Python, Ruby, Java	SenseTalk, Java, C#, Ruby
Record/ Play support	X	X	X		X		X
CI Support	X	X	X	X	X	X	X
Native/ Hybrid	X	X	X	X	X	X	X
Web	+/- (need to wrap the website into native app)	X (comes with wrapper)	+/- (need to wrap the website into native app)	X (comes with wrapper)	X (comes with wrapper)	X	X

[6] - http://appium.io/

[7] - http://xamarin.com/test-cloud

[8] - http://www.tricentis.com/tricentis-tosca-testsuite/tosca-mobile-plus/

[9] - https://www.cloudmonkeymobile.com/

[10] - http://www.sikuli.org/

[11] - http://www.testplant.com/eggplant/

[12] - http://mobilelabsinc.com/products/deviceconnect/

2.2.1 *Tuneup JS*

The main achievement of TuneupJS[13] 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[14] 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.

2.2.2 *mechanic.js*

mechanic.js[15] is a CSS-style selector engine for UIAutomation. It also allows accessing UIAElements and executing the commands with a shorter notation.

3. tTAP EXTENSION FOR APPLE UIAUTOMATION

When doing the first proofs of concepts in UIAutomation we of course took a look at both of the previously mentioned extensions. 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. During the extensive test automation process it appeared that we need the different sets of commands to make our live easier. That is why we started to cut, rewrite, and extend Tuneup JS extension that resulted into new extension creation that we call tTap – target tap. The main reason for this title is that almost all actions within the extension are executed in absolute coordinates of the device while still operating on the UIAElements (UIView and UIViewController) level. The device (or simulator) is called *target* in UIAutomation context. The decision to work in absolute coordinates was made to overcome several issues that we will describe in a course of this Section. tTap extension uses the JavaScript test runner and assertion functions from Tuneup JS extension, as well as borrows some UIAutomation class extensions and the image comparison idea. It is worth mentioning that tTap extension is distributed under MIT license[16].

3.1 Solution Details

tTap is not only the notation extension for UIAutomation. It is the whole test framework model that comes with the template to make the test domain-specific language (DSL) for the new app more quickly and in a more convenient way.

The test framework consists from the DSL and tests themselves. DSL consists from UI libraries and actions sets. Each UI library and action set in most cases are the separate files.

Single UI library describes all main UI elements of the screen or of the screen part (e.g. toolbar, menu, etc.). For this purpose the accessibility identifier is set per each element. The accessibility labels should not be used for this purpose, because they should be different per each app language. Accessibility API of iOS uses labels to navigate within the system and apps using the voice control, while they still can be accessed by UIAutomation. Accessibility identifiers should be unique in the most cases, but sometimes they can be the same for some UI elements groups that behave in the similar way. UI library is a JavaScript file containing the constants with accessibility identifiers.

Action set is a JavaScript object that extends the specific Screen object of the extension. Screen object gives the shorter notation to access target, app, window, navigation bar, toolbar, or keyboard elements. Action set consists from functions that search and return UIAElements by the accessibility identifiers and do the definite action with these UIAElements.

The modified test runner from Tuneup JS is used to run the test. Tests follow the unit tests style. 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 introduced the possibility to ignore some tests in general or not to run some tests if some another test has failed.

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 [5].

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 Figure 2. This solves the following:

- Sometimes, UIAutomation does gesture on the wrong coordinates if the command is called exactly from the UIAElement. It sometimes occurs with 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.

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.

The authors have improved the image comparison solution that comes with Tuneup JS. It used to fail when there were some tens of files on the desktop, because the screenshots temporary are stored there. We also have rewritten it to do the comparison of images with some delta. 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.

The extension allows switching on/ off the internet. The Link Conditioner that is the part of Xcode developer tools is used for that purpose. It is called using the commands written in Apple Script. The internet from the machine where

---

[13] - http://www.tuneupjs.org/

[14] - http://www.imagemagick.org/

[15] - http://www.cozykozy.com/mechanicjs/

[16] - http://opensource.org/licenses/MIT

automated tests are run should be wirelessly shared with the device under test to use this feature.

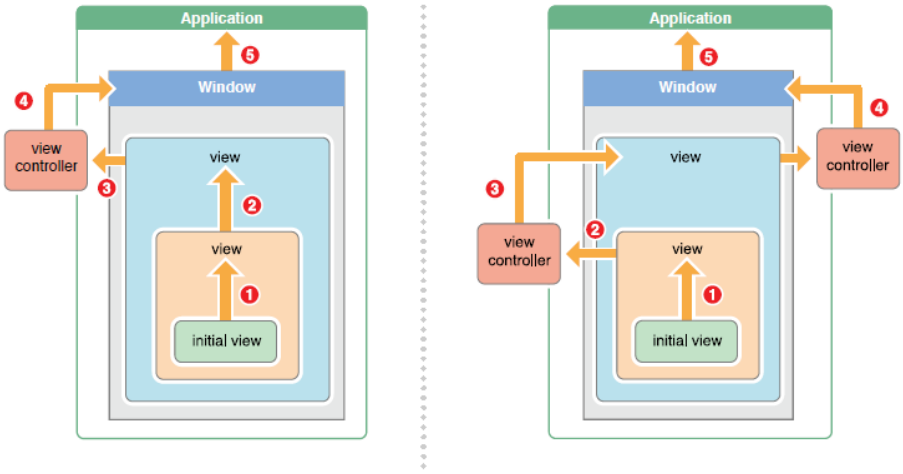


Figure 2. The examples of responder chain in iOS [3].

3.2 Practical Usage Experience

3.2.1 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 8, but pinch to zoom inside the scroll view still does not work on a simulator. There were also some cases when buttons on the system modal windows, e.g. mail controller responded to the automation only after they were tapped manually for the first time. Of course, this is not acceptable. It could be that this issue with buttons appeared again due to the usage of Xamarin cross-platform solution, but the same works properly on a device.

We have tried to run the tests on simulator for nightly/ daily builds from the continuous integration (CI) server. Of course, it is possible that some other builds are executed on the build machine where these tests were executed. That ended into random test failures, or into the failures that we could not repeat when running the same tests on the simulator locally. It appears that simulator speed can differ significantly during the test execution on the build machine under an additional load. Even special hooks that we create for being sure that element is present on the screen did not help. Another issue is related to keyboard. We have adjusted the default inter key delay of the keyboard till 0.2 seconds that makes typing more robust, because it constantly failed on the simulator of the build machine when switching between the keyboard types (e.g. numeric, capital letters). But in the cases of higher load or when build machine was not restarted for a long time this still did not help. But this trick is still applicable for the device, because the test typing can fail there as well with the default delay of 0.03 seconds.

Another thing 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 occurs in the freezing manner, while the same works OK on the device. Another example could be the difference in precision of epsilon value on different architectures. Epsilon is the smallest positive float value.[2, 7] 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.

The last, but not least thing to be mentioned for the device vs. simulator battle is that app can crash on the device easier than on the simulator due to the memory management things.

Finally, we have configured the Jenkins[17] server on the separate build machine with extended test reporting, connected multiple devices to it and run tests from it using UIAutomation command line directives. It is worth mentioning, that UIAutomation speed decreases during the long test runs and it can fail unexpectedly at the end when they are executed through the UI of the tool, while we have not expected such unexpected failures when running the tests from the command line.

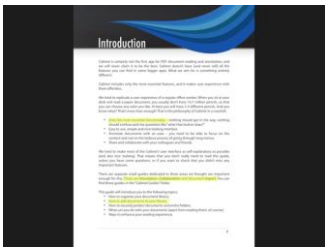
3.2.2 Image comparison

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 that. The example is depicted in Figure 3.
- 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 Figures 4 and 5.

[17] - http://jenkins-ci.org/

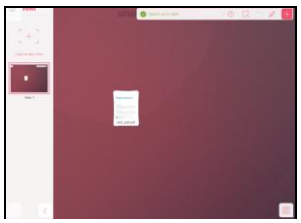
3a. Start drawing annotations



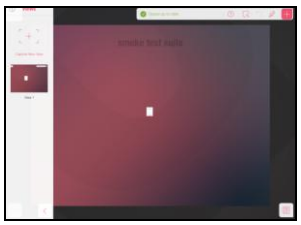
3b. Drawn annotations

3c. Comparison result

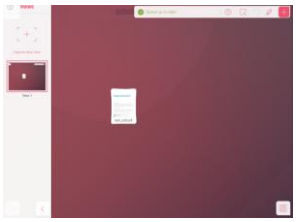
Figure 3. Image comparison example of OpenGL activities.



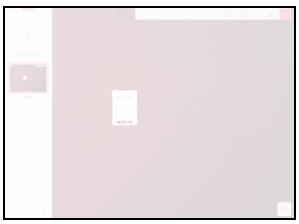
4a. Original bookmarked viewport.



4b. Zoomed out view.

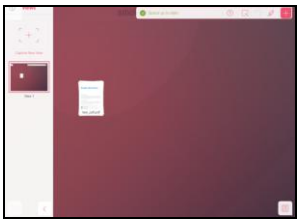


4c. View after navigating via bookmark.

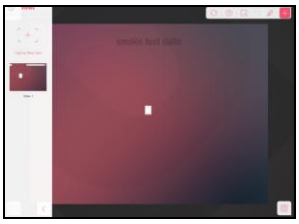


4d. Comparison result of 1 and 2. The difference is shown in bright red. Small delta is allowed.

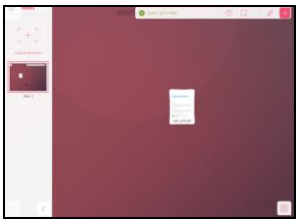
Figure 4. Image comparison passed test example of viewport bookmark functionality.



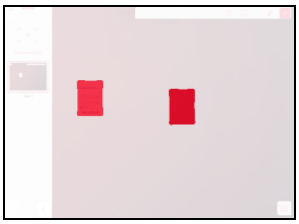
5a. Original bookmarked viewport.



5b. Zoomed out view.

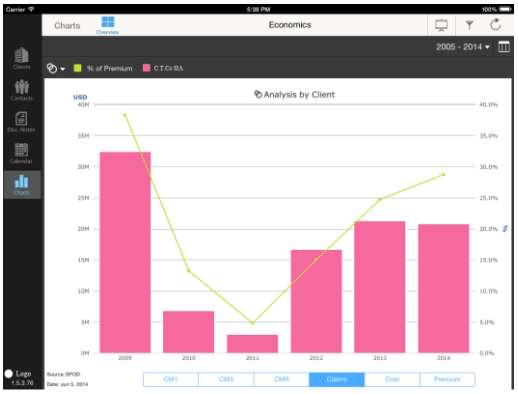


5c. View after navigating via bookmark.

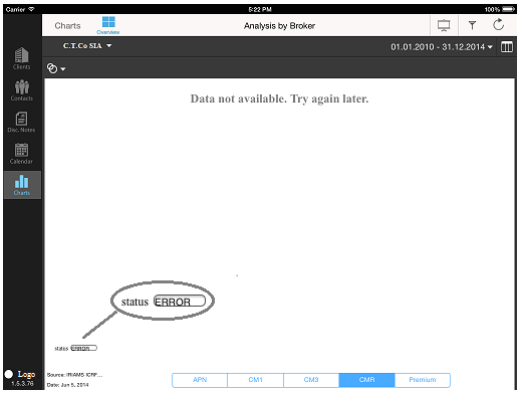


5d. Comparison result of 1 and 2. The difference is shown in bright red.

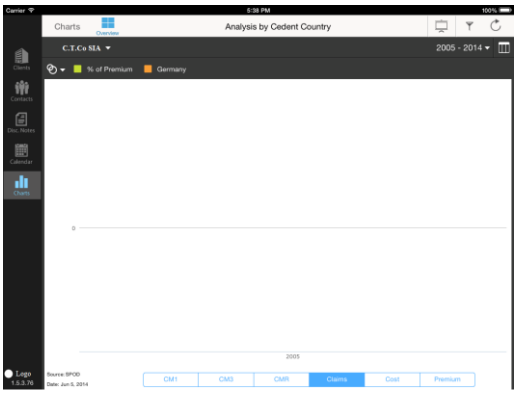
Figure 5. Image comparison failed test example of viewport bookmark functionality.



6a. App overview with properly drawn chart.



6b. Example when chart is not drawn, but app's logic or HighCharts library has caught the error.



6c. 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.

Figure 6. Image comparison testing: the example of comparison with background technique.

The image comparison can 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 clients and other filters using the HighCharts[18] JavaScript library. The app is integrated with the server through REST JSON services that convert the data from database to the proper 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 to catch more than 10 bug 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 Figure 5. The test script was iterating through the different clients, options, filters, etc.

4. CONSLUSIONS AND IMPLICATIONS FOR FUTURE WORK

In a course of the study the authors have divided the mobile automation tools that can automate iOS apps into the following categories: OEM vs. 3rd party, cross-platform vs. single-platform, wrappers vs. library integration in the app source code; API-based vs. image-based, etc. The weak points of each category have been described as well. They all are related to the additional layers that can break in comparison to OEM solution from Apple called UIAutomation. Taking into account that most of the apps in our company are created for iOS, and there is no need for the cross-platform support currently, we have decided to stick to UIAutomation solution because of the less number of weak points in comparison to the other solutions. When choosing the right tool we have acknowledged the limited debugging capabilities of this tool due to the own, non standard JavaScript environment where tests are executed.

[18] - http://www.highcharts.com/

While automating the UI tests for iOS apps we have created our own tTap extension for Apple UIAutomation that initially is based on Tuneup JS extension. tTap is distributed under MIT license. It comes with the model template for DSL creation when starting your own test automation project. We have described the importance of the accessibility identifiers that should be used to describe the objects in UI libraries. Our extension also includes the function for recursive search of UIAElement within the UI tree (while UIAutomation itself searches only within the single node of the tree) and several other convenient notations to shorten the test scripts.

It also overcomes several issues that the authors have faced during the extensive test automation process like some button on the system window cannot be tapped or some keyboard key cannot be pressed. This is done by doing the actions on a target level in translated absolute coordinates of UIAElement and by setting the proper inter key delay to allow automation to switch between the different keyboards types like numbers, capital letters, etc. The keyboard automation improvements work in all circumstances on the device when running the tests from the command line, and in most cases on the simulator of not overloaded build machine.

We have shared our practical experience convincing why real tests should run on the real device while simulator in most cases could be used only as a test design tool. The arguments are the following: pinch to zoom inside the scroll view does not work on simulator starting from iOS 7; tests in simulator randomly fail when are executed on the loaded machine; device uses ARM processor, while simulator uses x86 processor that just makes them to behave differently in some situations; it is easier to crash the app on the device if there are memory leaks than on simulator.

The authors have improved the image comparison functionality taken from Tuneup JS that is based on ImageMagic tool. The situations when image comparison is almost the only option available are described. They are: tests for elements created by OpenGl, e.g. drawing the annotations; tests for bookmarks of viewport of definite zoom and position; tests that compare the chart or table area with the background indicating when there are too little or no elements on the screen. Another improvement is a possibility to switch on/ off the internet or limit its speed through Network Link Conditioner directly from the test.

It is worth recalling that UI end to end tests in most cases should be limited to the happy path flows, basic create, read, update, delete (CRUD) functionality, navigation, and, probably, some corner cases of the special interest, because the maintenance effort is still much higher than for the tests from the lower levels. While we have achieved quite robust solution by using tTap extension functions and proper DSL model the maintenance effort is still quite high in comparison to unit or integration tests.

The authors have configured Jenkins CI server on a separate build machine for the only purpose of the scheduled and manually triggered automated UI tests execution on the real devices. Tests from other levels are executed during each build on another CI server. This test lab including the tests from the lower levels should be described in detail in some next publication, so that the reader could understand and apply the full test automation solution. The authors also plan to describe how UIAutomation with tTap extension is used for testing the stability of the apps.

5. ACKNOWLEDGMENTS

The authors thank "C.T.Co" SIA[19] for providing the ability to create and improve tTap extension while participating in complex real mobile projects.

6. REFERENCES

[1] Citrix Data Reveal New Global Trends in Consumer and Enterprise Mobility, 2015. Retrieved March 02, 2015, from Citrix: http://www.citrix.com/news/announcements/feb-2015/citrix-data-reveal-new-global-trends-in-consumer-and-enterprise-.html.

[2] Double.Epsilon Field, 2015. Retrieved March 02, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/system.double.epsilon(v=vs.110).aspx.

[3] Event Handling Guide for iOS, 2013. Retrieved March 02, 2015, from Apple Developer: https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/EventHandlingiPhoneOS.pdf

[4] Good Technology™ Mobility Index Report Q4 2013, 2013. Retrieved March 02, 2015, from Good Technology: https://media.good.com/documents/rpt-mobility-index-q413.pdf.

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

[6] Scott, A. Introducing the software testing ice-cream cone (anti-pattern). Retrieved March 02, 2015, from WatirMelon: http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/.

[7] Single.Epsilon Field, 2015. Retrieved March 02, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/system.single.epsilon(v=vs.110).aspx..

[8] UI Automation JavaScript Reference for iOS, 2012. Retrieved March 02, 2015, from Apple Developer: https://developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAutomationRef/_index.htm

[19] - http://ctco.lv/