This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

Meeting Quality Standards for Mobile Application Development in Businesses: A Framework for Cross-Platform Testing

Abstract

How do you test the same application developed for multiple mobile platforms in an effective way? Companies offering apps have to develop the same features across several platforms in order to reach the majority of potential users. However, verifying that these apps work as intended across a set of heterogeneous devices and operating systems is not trivial. Manual testing can be performed, but this is time consuming, repetitive and error-prone. Automated tools exist through frameworks such as Frank and Robotium; however, they lack the possibility to run repeated tests across multiple heterogeneous devices. This article presents an extensible architecture and conceptual prototype that showcase and combines parallel cross-platform test execution with performance measurements. In so doing, this work contributes to a quality-assurance process by automating parts of a regression test for mobile cross-platform applications.

1. Introduction

Software testing is widely used as a method to improve quality and reduce risk [1]. However, due to the complexity in software it is considered infeasible to prevent and find all possible defects within the time window and budget of a common software development project [2]. Software testing is therefore focused on the most risk-reducing techniques within the constraints given.

To understand some of the challenges related to testing of mobile applications, it is useful to first summarize the different approaches for developing them. Firstly, there are native applications, namely those written in the respective platform's native programming language. For iOS apps this means Objective-C, for Android it is Java and for Windows Phone it is C#. Applications are distributed using their platform's respective app store, and some require an approval process that can take several days before an application is published or updated. If an application crashes or contain bugs, a user of the respective application's app store may post a negative rating. Even if the developer were able to fix the bug immediately, it would be subject to a new review process on iOS and WP7 that could again take up several more days. Moreover, when the fix is published one cannot force the users to upgrade their applications. The importance of testing native applications is therefore obvious. The challenging part of such testing is the variety and diversity the number of devices and OS versions in use. This leads to the testing process being time-consuming and it is difficult (if not outright impossible) to cover all versions and variations of exiting devices.

Secondly, there are mobile web applications. These applications are mainly developed using the same tools and languages as regular web pages, but with a touch-friendly user interface. Although web applications allow developers to quickly deliver apps to several platforms, they come with similar problems as their desktop counterparts — cross-browser compatibility. Testing can be difficult due to cross-browser issues on a variety of devices. On the other hand, several web testing tools such as Selenium, JSTestDriver and Buster.js attempt to remedy this with cross-browser test support.

Automated testing attempts to reduce the amount of manual work in testing. It can, however, never replace manual testing and is most often used for unit and regression testing [3, 4]. Additionally, Berner et al. [3] list time-consuming development of tests, neglected test environment, repetitive tests and maintainability as common arguments against automated testing.

Given the diversity of mobile platforms, devices and development techniques, it seems highly plausible that mobile applications could benefit from automated testing. Automated testing attempts to reduce the amount of manual work in testing, and frameworks for GUI testing native applications on iOS and Android already exist. However, to the best of our knowledge no test framework is able to run the same test against an application developed for multiple mobile platforms. Given that iOS and Android, combined,

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

dominate the mobile operating system market, this then leads to our research question: *How can an extensible cross-platform testing framework be constructed in order to automate GUI testing of mobile applications on iOS and Android?*

2. Background

In the earliest days of software engineering, testing was reactive and revolved around fixing defects after they occurred by debugging the software [5]. Since 1988 the approach has been prevention-oriented and this proactive attitude is still considered the right approach [1]. It is better and cheaper to find problems early rather than finding them late or in production [6]. Still, it is considered infeasible to prevent and find all possible defects within the time and budget of a common software development project [2]. This is also reflected in the IEEE Computer Society's definition of software testing [1].

A study on Android fragmentation revealed 3997 distinct device models [7]. If we combine this with the different OS versions on Android, we clearly see that testing an application on all combinations of devices and operating systems is also practically infeasible. Other key issues in software testing include selecting a suitable set of test cases, optimizing towards the most efficient ways to test, and developing software that is testable. Software testing is also done for different objectives. A subset of these includes acceptance testing, performance testing and regression testing. In acceptance testing the software is tested to verify that the customer's major functional and non-functional requirements are met [8].

Regression testing is performed on existing software to ensure previous tests still pass after modifications have been made [1]. Ideally, all parts of the software would be retested, but time and budget require prioritizing the features to retest [2]. Thus, with the aforementioned landscape of mobile devices and versions, regression testing can be particularly challenging.

Performance testing attempts to discover how well a test subject performs under different conditions and load, for instance how many users your app is able to handle simultaneously, or how many devices a testing tool can handle at once.

Testing can also be used as a part of software design, usually on the level of unit and integration testing. Accordingly, test-driven development (TDD) involves writing a small test first, implementing code to make it pass, and refactoring that code to reach production quality [9]. Since TDD's focus is on making small and fast increments between test and implementation, a testing tool that supports this fast feedback cycle is desirable. Moreover, it is debatable whether TDD leads to improved reuse

2.1. Automated Testing

To reduce the manual labor involved in testing, automated testing can be employed at different levels and objectives. It is a widely used industry practice, mostly for unit and regression testing [4]. Libraries such as JUnit (Java)[1], Mocha (JavaScript)[2] and OCUnit3 (Objective-C)[3] can be used to write automated tests. These are typically used to verify that components behave correctly on a unit or integration test level. Another example is Selenium[4]. It is a browser automation framework used to verify that a web application works as expected in a set of browsers (Firefox, IE, Safari, Opera, Chrome). It is able to run on multiple platforms and can be used for automated regression testing.

Although automated testing is widely used, it is often employed with unrealistic expectations, such as saving money on "unproductive" testing activities, time and testing resources [3]. Automated testing cannot replace manual testing, however: "With automated tests, the expert testers are freed from running the same boring regression test suite over and over again and more resources are available for difficult tasks" [3].

Moreover, a study performed by Kasurinen et al. [4] found that organizations only automated 26% of their test cases, suggesting that test automation was a demanding effort. They also found cases where automation was discarded on smaller projects, due to high start- up costs. Observations in Berner et al. [3] support this and found that maintaining the test scripts, test data and the test environment is hard, resulting in high maintenance costs. Also, tests must run frequently or they will not be maintained. This becomes a problem when they cannot run without a significant investment in fixing the outdated tests.

Berner et al. [3] also found that automated tools usually focus on the test execution itself. However, installation, configuration and reporting are often neglected, even though doing so can significantly

[1] http://junit.org/

[2] http://mochajs.org/

[3] https://developer.apple.com

[4] http://www.seleniumhq.org/

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

reduce the total time spent on testing. This is very relevant in the context of testing mobile applications, where a diverse set of devices, OS versions, and applications and build configurations must be maintained for a cross-platform application.

As mentioned, automated testing is mostly used for unit and regression testing [4]. A more specific area of automated testing is automated GUI testing, where the input and outputs of a graphical user interface can be automated. This can be used for regression testing, and is discussed next.

2.2. Automated GUI Testing

Automated regression testing of GUIs has been described as a "GUI smoke test" [10]. The principle is that a build server should be able to verify that the major parts of an application still work after modifications have been made. Ideally, it should be able to run on multiple machines (in parallel) to reduce time spent on testing. A common argument against automated GUI testing in general is that the tests can have false positives and be expensive to maintain [10].

In Adamoli et al. [11], an extensive survey of prior techniques for GUI testing was performed. Of the 50 surveyed papers, 18 used a technique called "record & playback", in which the tester performs actions on the GUI while the tool records these actions for playback later. Depending on how the tool is implemented, a problem with this approach is change. Just moving a button can render the test case useless. The remaining papers in Adamoli et al. [11] used techniques not strictly bound to event sequences, called model-based testing. A model intends to abstract the event sequences away so steps can be reordered, inserted and deleted with minimal effort [12]. This can remedy some of the maintenance costs associated with automated testing.

A concrete example can be found in Jaaskelainen et al. [12]. Here “system API” is described as a method, meaning that the mobile application exposes an endpoint capable of answering question regarding the current system state. The authors exclude the GUI from this method, but similar approaches also exercising the GUI exist and are described as “keyword and action word” testing [13]. All these build on the concept of model-based testing.

A less popular alternative is assertion with images, according to which screenshots are taken of the application during tests and compared to the “expected image” a test designer supplied beforehand. Kwon and Hwang [14] developed a testing tool to easily model the flow with expected screens and ran these against a device. Maintainability is the main problem with this solution, as changing a color used in many places requires updating all the images to their new version [15]. Also, with the frequent use of animation and platform-specific GUI components on iOS, Android and Windows Phone today, one would need three set of “expected images” or sophisticated algorithms to cater for all variations.

Based on the literature, it thus seems that a model-based GUI testing is the most flexible approach. To gain further insight and background information the abstraction was increased and focus moved to abstraction layer.

2.3. Abstractions as a Key for Testing

Abstractions are employed in software engineering to reduce complexity. For cross-platform testing a loose coupling to the underlying platform is required to abstract the different platforms implementation away. This may be achieved by using a language that is not tied to a platform or programming language.

A challenge in cross-platform testing is identifying User Interface (UI) elements across applications on different platforms. Adamoli et al. [11] state that the problem is present even in applications without cross-platform support. One reason for this is that capture and replay tools often store a very specific reference to the targeted element making it fragile to modifications later. Another reason described by Adamoli et al. [11] is the “temporal synchronization problem”, which can occur in testing applications depending on animations and clocks, and may result in timing issues. This can render the element invisible or disabled and thus prone to failing the test if the testing tool doesn't account for these timing issues. A related approach found in Matos and Sousa's [16] work is capable of generating a mocked user interface along with functional tests based on use case models. Use case scenarios are written in a “controlled natural language”, i.e. English with strict language semantics. This enables the non-programmers to understand, and even write the test without any programming skills.

A similar approach is found in Cucumber[5]. Cucumber is widely recognized in the Ruby community and several books are written on it. The requirements can be specified in a neutral language called Gherkhin[6]. A strength of Cucumber and Gherkin is that the tests can be written in any format and language as user stories

[5] https://cucumber.io/

[6] https://github.com/cucumber/cucumber/wiki/Gherkin

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

and follow the *Given...When...Then* format used in Cucumber. A mapping between test description and the actual functions can then be written next and is used to relate this to developer environments.
In summary, testing an application across all platforms and supported devices is time consuming and error-prone when done manually. While there exist automated GUI testing tools for iOS and Android, research is lacking on whether these platforms can be test-driven simultaneously with the same test and same tool. This is the precise focus of this article, which presents Mobilette, a framework for testing mobile cross-platform applications. Accordingly, the structure of the remainder of the paper is as follows: the methodology employed in our research is presented next; Section 4 then describes the Mobilette framework, while Section 5 details its evaluation results. Lastly, conclusions are drawn and opportunities for future work identified in Section 6.

3. Methodology

The methodology used in this project is the Design-science research paradigm, as described by Hevner et al. [17]. Design-science research was used to structure the process for investigating the research problem. It consisted of all the common phases from collecting objectives of a possible solution, to design, development, and evaluation. The application of Hevner et al.'s [17] guidelines are summarized in the following table.

Table 1. Design Research approach

Guideline	Research outcome
1. Design as an Artifact	An instantiation of a test framework as an artifact.
2. Problem Relevance	Relevance proven as outcome of the background section.
3. Design Evaluation	Demonstrated via the test cases presented in the evaluation framework
4. Research Contributions	To the best of our knowledge, no similar framework exists. Thus the contribution will be an instantiated artifact, architecture and empirical data.
5. Research Rigor	Best practice in information systems SE was applied to the development of the artifact.
6. Design as a Search Process	An iterative process followed during the development of the artifact.
7. Communication of Research	Communication through research article

4. Mobilette Framework

The research question posed at the outset of this article is investigated – and subsequently answered – through the development of Mobilette - a test framework for cross-platform mobile applications – which we now proceed to describe.

4.1. Functionality

The framework consists of four main parts. The server is the heart of Mobilette. It maintains a list of all connected devices, and sends commands and receives responses from these. It also includes the necessary components to build a test framework on top of these devices and commands. The Android Robotium driver

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

is included in an Android application to make it communicate with the server. It will parse commands from the server and translate them to the underlying test framework (Robotium). Third, the iOS Frank driver is included in an iOS application to make it communicate with the server. It will parse commands from the server and translates them to the underlying test framework (Frank). Finally, the client is the user interface of Mobilette. It has an "interactive mode" and an automated mode. The components are modularized into individual pieces.

When first starting the Mobilette server, it enables the deployment of applications to each platform and then runs the Mobilette client to perform tests or execute commands (Figure 1.).

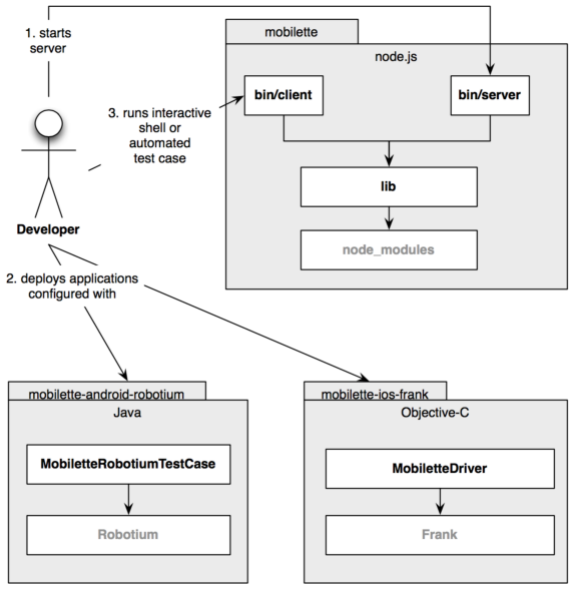

Figure 1 Mobilette components

4.2. Architecture

As described earlier, some platform specific tools already exist, but there is a lack of cross-platform ones. The first architectural choice was therefore to integrate with existing test frameworks to support mainstream development branches. The frameworks need to be able to work with both physical devices and virtual simulators or emulator. This is essential, since both physical and virtual devices are used during the development of a mobile application. Further, the frameworks should be well established and support a broad amount of commands. As the underlying test frameworks performs the actual GUI manipulation and instrumentation, the challenge for Mobilette becomes to integrate with these frameworks in a platform-independent way. The solution became what Mobilette calls a "driver", and the concept is depicted in Figure 2.

A driver is responsible for two things. Firstly, it abstracts the underlying test framework away by implementing a common interface. This is the topic of this section. Secondly, a driver is responsible for registering and keeping in touch with the server at regular intervals, using “heartbeats”.

A driver implementation for each test framework is found both in Mobilette's server and on the application under test. The server translates a generic command such as "touch" into a format that the underlying test framework supports and transfers it to the remote interface. If the underlying framework has a remote interface it will be delivered directly to it. Otherwise the remote interface will be created in Mobilette's driver to communicate with the underlying test framework. This is further illustrated below in Figure 2.

The specific language of the platforms, i.e. Android and iOS, are interpreted and converted from the high-level test language by the Mobilette driver modules. These modules incorporate the integration of the language specific interpreter, Robotium and Frank. By writing the tests in a high level language, we are able to abstract away from platform and language details, focusing on core test outcomes. Furthermore, this facilitates user-centered design of tests and the possibility to include non-technical people in writing and assessing tests. On the backend, the controlling server instance is written in NodeJS, which is a standard, modular JavaScript based implementation able to run on all operating systems. Moreover, the development roadmap of the framework indicates secure maintenance and updates for the coming years.

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

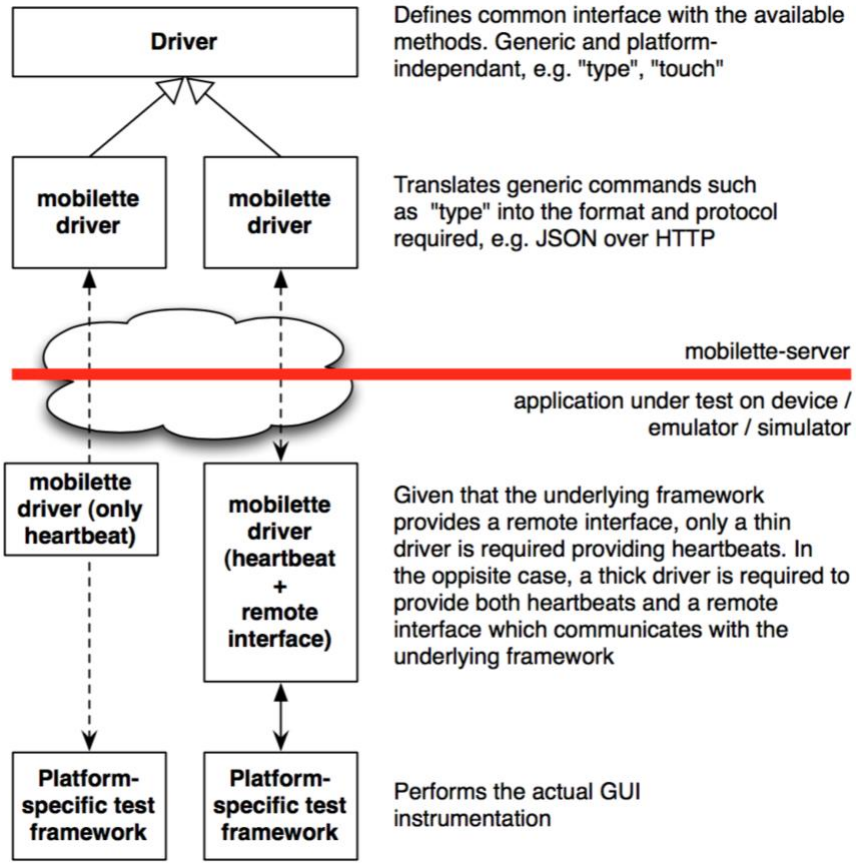

Figure 2 High Level Mobilette Architecture

4.3. Communication through Heterogeneous Architectures

Based on client server architectures, the implementation of Mobilette is centered on controlling real time events. Remote controlling devices in real time requires the devices to have an open channel that listens for incoming requests. This channel should be built upon a protocol that both a regular computer acting as a test host and a mobile device supports. The obvious choice is TCP/IP, which is supported by modern mobile operating systems. Transporting data could then be done over a socket opened over TCP/IP or via a higher-level protocol building on TCP/IP. Mobilette choses the latter and uses HTTP as its application protocol.

HTTP is widely used and well supported on all relevant platforms and technologies. It is request-response based and fits well with Mobilette's need to send commands and receive responses from devices. However, HTTP does not maintain a bi-directional connection in which both the server and client can communicate freely in both directions. This is a limitation for real-time applications such as chat and collaborative software. WebSockets[7] is a protocol that addresses this issue and is capable of bi-directional, full-duplex connections. Although WebSockets is a more responsive protocol, it is not as widely supported as HTTP and may be deprecated in cases where HTTP solves the same problem. Based on this, our communication takes place between server and drivers over HTTP, and between server and client over WebSockets. This is further illustrated in Figure 3.

This architectural set up allows for scalable device management. By having the clients loosely coupled from the architecture and the drivers for supporting the different platforms included, it is possible to add any number of devices desired. Further, this technique allows for scalable maintenance of the platform frameworks. For instance, when new iOS or Android versions are released the maintenance of the Mobilette test framework is limited only to the driver

[7] https://www.websocket.org/

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

implementation details in Frank and Robotium, respectively.

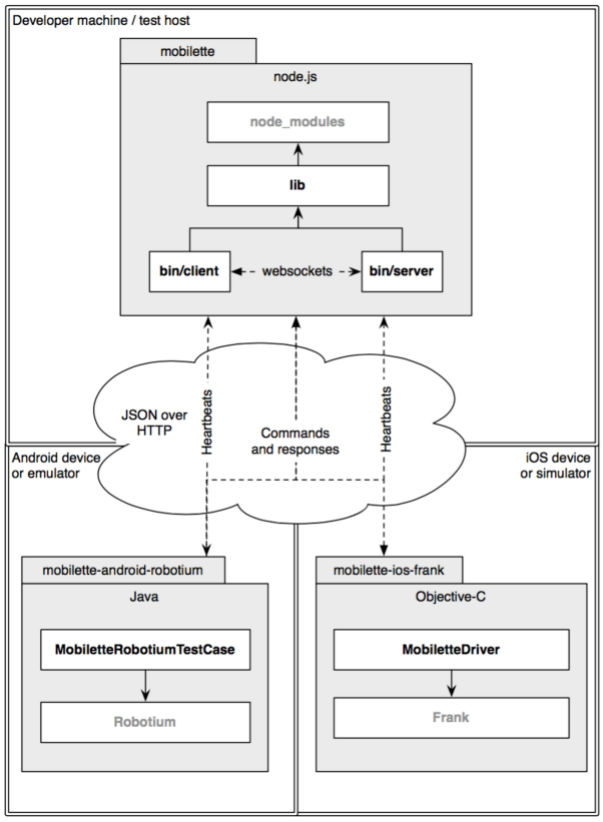

Figure 3 Deployment Diagram of Mobilette

For real time communication Node.js is chosen as it runs on Windows, Linux and Mac OS X. This is a benefit as developers should be able to use Mobilette on the platform they are developing applications for. Mobilette could have been written in any server-side cross-platform language such as Java, but Node's fast event-driven and non-blocking I/O makes it particularly attractive for Mobilette, as it fits with Node's description of a data-intensive real-time application that run across distributed devices. Node applications are also written in JavaScript, which give the benefit of future, web-based, clients of Mobilette the option to reuse models on the server if desirable

Before any commands can be sent, the server needs a list of devices to send the commands to. This list may update at runtime as devices connect and disconnect. The Device Manager handles this, which is a singleton. A device will contact the server to register itself for testing. The Device Manager will assign an ID to the device and put any meta information it received from the device into a registry of registered devices. This meta information will contain the operating system it runs on, which Mobilette driver it uses, the IP-address it can be contacted on, its screen size and more. If the device is not heard from again within a few seconds, it is considered dead and deregistered by the DeviceManager. To stay alive, the driver-part on the device will send heartbeats every second with the assigned ID and meta-information. This way, the server can maintain a fairly accurate list of available devices, without actually having an open connection to them.

Heartbeats are inspired from test drivers. Under this framework, browsers such as Chrome, Firefox and Internet Explorer can act as devices where the same test is run to ensure that behavior is consistent between multiple browsers. Browsers will continuously report back to the server that they are alive and what they are currently engaged in, such as running a command. In this respect, Mobilette is somewhat similar to the JSTestDriver[8]. Both start a server that is capable of receiving connections from devices or browsers they can later command. Heartbeats are sent to let the server know who they are capable of controlling.

4.4. Evaluation

In order to provide reproducible results, a stable and reliable test environment has to be established. Figure 4 provides a graphical overview of all key components involved.

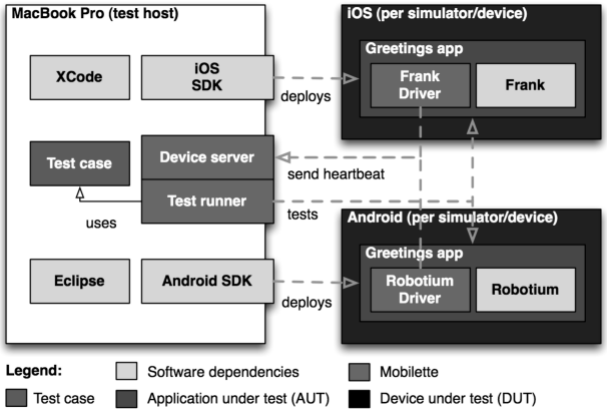

Figure 4 Test set-up

A specification of the hardware used as a test host and devices under test is listed below in Table 2. The tests could run on a wider set of devices, but are distilled down to the minimum amount of devices needed to

[8] https://code.google.com/p/js-test-driver/

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

answer the research question. The devices and host were connected on an isolated network in order to minimize interference from other network traffic.

Table 2 Device specification

	Host	iOS emu-lator	Android emulator	iOS device	Android device
Dev	MacBook Pro	(host)	(host)	iPhone 6	Galaxy 5
OS	OS X 10.10.3	iOS 8.3	Android 5	iOs 8.3	4.4.2
CPU	2.6 GHz i7		ARM emulated	ARM v8	Quad 2.5 GHz
RAM	16 GB	N/A host	1 GB	1 GB	2 GB

The required software used for testing was:

- XCode 4.2
- Eclipse 3.7
- Frank 0.8.14
- Robotium 3.2.2
- Device server (part of Mobilette)
- Test runner (part of Mobilette)

All software not required by the OS was terminated on the test host and the devices under test to minimize interference with performance and test results. Additionally, the applications under test were killed and restarted after each test to further minimize the possibility of the result.

Mobilette supports a subset of interaction elements in the UI such as touching elements, setting text and getting text. The tested application is shown in Figure 5 and tests all of these features. It is deployed on all devices under test and works in the following manner: when the user enters a name and clicks "Greet", the application should respond with "Hello <user>!"

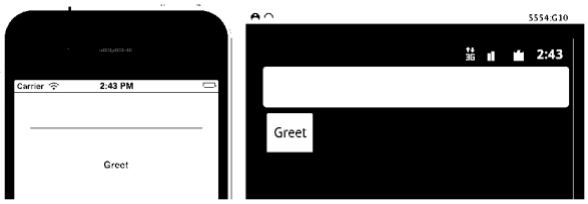

Figure 5: Application under test. iOS to the left, Android to the right

The artifact tests are divided into two broad categories, which will now be described: the *acceptance* test is a high-level test that verifies if the developed artifact works according to the main functionality and requirements [8]. It was performed by running the same test case multiple times, but with small changes (bugs) to the application under test (Greetings), causing the AUT's test case to fail if Mobilette works correctly. Secondly, the *scalability and performance test*'s purpose is to gather metrics on how scalable and fast Mobilette is in practice, which is an indication of its usefulness as an automated testing tool [10]. A performance test also falls under the category of "measurement techniques" in the evaluation of software architectures [18].

Thus, the test investigated how well it handles multiple devices at the same time by monitoring response times for a test to complete. Any differences between Frank and Robotium, and between simulator, emulator and physical devices can be thus uncovered. The amount of test runs and metrics is derived from Adamoli et al.'s [11] approach to GUI performance testing.

5. Results

Table 3 shows the result of running each test in the acceptance test. All tests were successful under expected conditions.

Table 3: Acceptance test results.

	iOS emulator	Android emulator	iOS device	Android device
Test case	Pass	Pass	Pass	Pass
No text field	Pass	Pass	Pass	Pass
No button	Pass	Pass	Pass	Pass
No label	Pass	Pass	Pass	Pass
Fix bugs. Incremental deploy	Pass	Pass	Pass	Pass

Overall, the acceptance test proved that the main functional requirements were met, and that the acceptance test can be considered successful. Improvements can be made to how Mobilette handles unexpected situations such as screen locking, incoming phone calls and low battery warnings. This was not a part of the requirements, but should be considered in future versions.

Handling unexpected situations is device-specific and suitable for handling in Mobilette's drivers. Taking screen locking as an example: it may be possible to solve this by configuring the timeout manually on all devices, but this is not ideal and not always possible. For instance, iOS enterprise profile restrictions may restrict the timeout from being set longer than 5 minutes. Instead, Mobilette's driver may run code that prevents the screen from sleeping, in a similar way to how video players and games function.

Another observation is that while the tests were quick to run, deployment and configuration is time

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

consuming. This corresponds to Berner et al.'s [3] findings, suggesting that automated test frameworks should also focus on automating installation and configuration. Depending on the use case, the time to deploy applications to all devices may be too long. For instance, Test Driven Development (TDD) uses small code increments and fast feedback cycles between test and implementation [9]. When used as a tool for regression and acceptance testing the time to deploy is more acceptable. However, future versions of Mobilette could remove the manual deployment step by supporting automated deployment via the respective platform's SDK. Android has mature command line tools ready to perform this task. To the best of the our knowledge, the task is more difficult on iOS, where deployment is only available from the XCode IDE. It may be possible to perform this using "fruitstrap", a tool that reverse-engineers Apple's private API for deploying to devices from the command line [19].

In respect of the scalability and performance test, in Figure 6 we compare response times between different test-run configurations. A test case is first executed on each device separately. Then the simulator and emulator would run in parallel, followed by the physical devices in parallel and, finally, all four devices in parallel.

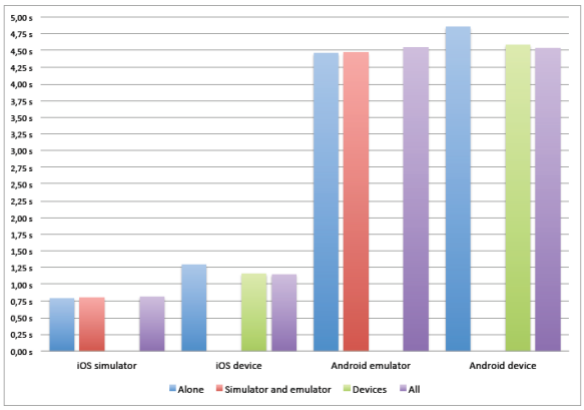

Figure 6: Performance test results

Results highlight nearly no performance degradation between running tests on a device alone or in parallel with all other devices. The total time required to run a test case is limited to the slowest device. This suggests that even more devices may participate in parallel test runs. Based on the current performance degradation, it is more likely that other factors such as SDK limitations and available USB ports will limit the amount of devices. Moreover, being able to run the test on many devices in parallel means that less time is required to wait on test results [10]. On the other hand, if Mobilette is only used towards the end of a new release and not continuously during development, this may be of less importance to the user.

The results also show that testing on iOS is four times faster than on Android. This was not investigated further in our current work, but may be a result of one or more factors:

1. iOS performs better than Android in testing

2. Frank performs better than Robotium —and related to this: launching a test with a test session attached to IDEA (Robotium) is slower than launching the application detached from the IDE (Frank).

Another observation is that the Android emulator and device performs nearly identical with averages of 4.49 and 4.65 seconds. The relative difference is larger on iOS, with averages of 0.79 (simulator) and 1.2 seconds (device). This is expected as the Android emulator actually emulates the ARM architecture on a device. On iOS the simulator runs on a i386 architecture while the device runs on ARM.

The differences between the devices can be measured in seconds and are of little practical importance to a tester running a regression test. On the other hand, it may be of more interest to a tester using Mobilette as a TDD-tool [9].

6. Conclusions and Future Work

Existing research and tools point towards the need for a tool such as Mobilette. Accordingly, this paper has shown how a cross-platform GUI test framework for mobile devices can be developed and be open for extension. In so doing, the answer to the research question posed at the outset of this paper is a positive one.

Scalability and performance tests run on Mobilette found that tests can run in parallel on multiple devices with little to no performance degradation. Thus, the time required to run a test was the time spent by the slowest device. This suggests that a regression test on all relevant devices can be performed in parallel without performance issues. Mobilette's evaluation also highlighted that, while the tests themselves are fast to execute, building and deploying to a set of devices is time consuming. This is consistent with observations in [3] for traditional GUI testing. These findings indicate

This article has been accepted for publication in a future issue of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/HICSS.2016.706, Proceedings of the Annual Hawaii International Conference on System Sciences

that a cross-platform GUI testing framework is suitable for regression testing, but too slow to be used for test-driven development

A natural continuation of the work described in this paper would be to evaluate the artifact in a real-world environment —ideally on different projects to gather data on its usefulness and performance in practice.

It should be noted that the experiments were done in a controlled environment with a very basic application under test. This was done to maintain focus on the core challenges in cross-platform testing. Mobilette's architecture is designed to be extensible. More complex applications must add support for additional commands and better error handling in unexpected situations. Additionally, compatibility with corporate network configurations and potential firewall issues should be investigated.

Currently, UI elements need to be laid out similarly across platforms. Future research could also investigate how applications with multiple screens and different interface paradigms can be supported.

Last but not least, future research could also implement and investigate the benefits of adding support for hybrid and web-based applications. A tool with cross-platform support for testing native, hybrid and web-applications could be used in all types of modern mobile application development.

7. References

[1] IEEE Computer Society (2004). Software Engineering Body of Knowledge (SWEBOK), EUA. Available: http://www.computer.org/web/swebok [Accessed: 12-05-2015].

[2] Whittaker, J. (2000). What is software testing? and why is it so hard?, IEEE Software 17(1): 70–79.

[3] Berner, S., Weber, R. & Keller, R. K. (2005). Observations and lessons learned from automated testing, Proceedings of the 27th international conference on Software engineering, ICSE '05, ACM, New York, NY, USA, p. 571–579.

[4] Kasurinen, J., Taipale, O. & Smolander, K. (2010). Software test automation in practice:
empirical observations, Advances in Software Engineering 2010: 4:1–4:13.

[5] Gelperin, D. & Hetzel, B. (1988). The growth of software testing, Communications of the ACM 31(6): 687–695.

[6] Adrion, W. R., Branstad, M. A. & Cherniavsky, J. C. (1982). Validation, verification, and testing of computer software, ACM Computing Surveys 14(2): 159–192.

[7] OpenSignal (2012). Android fragmentation visualized. Available: http://opensignal.com/reports/fragmentation.php [Accessed: 22-04-2015].

[8] Hsia, P., Kung, D. & Sell, C. (1997). Software requirements and acceptance testing, Annals
of Software Engineering 3(1): 291–317.

[9] Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M. & Erdogmus, H. (2010). What do we know about test-driven development?, IEEE Software 27(6): 16–19.

[10] Memon, A. & Xie, Q. (2005). Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software, IEEE Transactions on Software Engineering 31(10): 884 – 896.

[11] Adamoli, A., Zaparanuks, D., Jovic, M. & Hauswirth, M. (2011). Automated gui perfor- mance testing, Software Quality Journal 19: 801–839.

[12] Jaaskelainen, A., Katara, M., Kervinen, A., Maunumaa, M., Paakkonen, T., Takala, T. & Virtanen, H. (2009). Automatic GUI test generation for smartphone applications - an evaluation, Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, p. 112–122.

[13] Hwang, S. & Chae, H. (2008). Design & implementation of mobile GUI testing tool, Convergence and Hybrid Information Technology, 2008. ICHIT '08. International Conference on, p. 704–707.

[14] Kwon, O. & Hwang, S. (2008). Mobile GUI testing tool based on image flow, Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on, p. 508– 512.

[15] Memon, A. (2002). GUI testing: pitfalls and process, Computer 35(8): 87–88.

[16] Matos, E. C. B. & Sousa, T. C. (2010). From formal requirements to automated web testing and prototyping, Innovations in Systems and Software Engineering 6(1-2): 163–169.

[17] Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design science in information systems research,
Management Information Systems Quarterly 28(1): 75–106.

[18] Dobrica, L. & Niemela, E. (2002). A survey on software architecture analysis methods, , IEEE Transactions on Software Engineering 28(7): 638–653.

[19] Hughes, G. (2012). fruitstrap. Available:
https://github.com/ghughes/fruitstrap [Accessed: 16-08-20