Equivalence Testing Mobile Apps

Mobile apps are often developed and then evolved on more than one mobile operating system. For the publisher of such apps, the problem is how to ensure equivalence of the product on the various platforms, in the sense of acting equivalently with respect to a test set. In this paper we present an approach that tackles this problem from two directions: architecture and testing. First we will explain the role and pitfalls of reference architectures. Then we will present our equivalence testing framework. It is based on mocking components of the target implementation. Instead of implementing mock components for all platforms, our approach is to implement these mocks only once and run them on a central server. On the tested target device, stub components are injected that forward to their server-based counterparts. Thus the exactly same test code is applied to all platform specific implementations.

Christoph Denzler, Daniel Kröni, Maxim Moschko | christoph.denzler@fhnw.ch

Mobile phone technology is evolving rapidly. The market is highly competitive and market share figures are changing year by year. New mobile operating systems are still emerging, and established ones have to be supported in different versions. This fragmentation forces application developers to program for different platforms and devices in order to reach a large customer base.

The term 'platform' refers to a mobile operating system (like iOS, Android, Symbian, etc.) and their application programming interfaces (API). Adapting an application to different platforms (porting) confronts the programmer with a variety of abilities (camera resolution, transmission rates, multi-threading etc.), interfaces (wireless communications abilities, sensors and user interfaces), programming languages (Objective-C, Java, C, C#, JavaScript) and concepts (multi- versus single-tasking, reference counting versus garbage collection or user interaction).

We faced fragmentation when we were asked to support software development for a startup company. The application dealt with calling a security service in threatening situations, and it required access to a camera, a GPS, the phone and the network. As the application will be hopefully rarely used, it should provide a familiar user interface, i.e. the user interface should not distract the user from sending an alarm.

These requirements (and some others such as memory footprint) ruled out the two currently established possibilities to build cross platform applications (see [2]): Native cross-platform frameworks, such as PhoneGap [7], and mobile web application frameworks, such as Sencha Touch [8]. In our case the problem with mobile web applications was the limited access to device features (camera, phone). A native cross-platform framework was not an option because it introduces a dependency to the framework provider: only those platforms can be reached that are supported by it.

As listed in [1] there are many ways to cope with fragmentation. However manually implementing an application for different platforms remains a strong option. It is often the most pragmatic way to satisfy all of the above mentioned requirements and to deal with the different partners who implement the different ports and who are not able or willing to give up their development processes. We decided to leave as is the development of the application on the different platforms but surround it with our approach consisting of architecture and tests.

In this paper we will present a process and the tools to pragmatically support multi-platform development by testing equivalence of the specific implementations. Our approach embraces the differences between platforms but does not hide their individualities.

The paper is structured as follows: Section *Overview* provides an general view of our equivalence testing approach. In section *Abstract Architecture* we show how a common architecture can be defined and we hint at some pitfalls to avoid. Section *Equivalance Testing Internals* conceptually explains how our testing infrastructure works. Finally, section *Case Study*, using a simple example, illustrates our approach and shows how tests are written to verify the equivalence of the different implementations. Comments on related work, conclusions and future directions complete this paper.

Overview

The decision which mobile platforms will finally be supported depends on the availability of features vital to the application, e.g. the availability of certain sensors such as a GPS receiver or a camera. Regardless of the platforms chosen they should not influence the applications requirements. These are specified only once and describe what the application should do. Implementati-

on details should not be part of a requirements specification, be it a functional or non-functional requirement. For example an application that manages notes needs the possibility to persist them – regardless of the platform on which it is finally implemented. Typically such requirements do not change between platforms.

Our approach evolves from a platform-independent architecture to which all platform specific implementations must adhere. This abstract architecture defines a structural contract between components. It is then enhanced by a set of integration tests which check the behavior of the system.

Once device specific implementations have been developed and tested separately, there is no easy way to ensure that the different test sets will be consistent. Thus, it is not possible to check whether or not an implementation fulfills the common requirements. Moreover, changing requirements demand implementation adaptions and test code adaptions for each platform. This is time consuming, demotivating and above all, error prone.

We introduce equivalence tests which run against the abstract architecture without consideration of any concrete platform. They are written only once and are executed on the server. Remote method invocation is used to invoke functionality on the mobile device. Our approach does not deploy real mock components on the mobile device. Instead lightweight proxy objects are injected during test setup. They are responsible for the forwarding of any interaction to their server-based counterparts which are implementations of the mocking objects. Our method uses a variety of approaches described in [1]. At first, manually implement the application on different platforms. Then, write test and mock components which will be abstracted from the target platforms by a remote method forwarding scheme. The necessary proxy objects are generated for each platform.

Abstract Architectures (AA)

In order to test different platform-specific implementations against a common test set, it is essential to establish a common basis. This basis is provided in form of an abstract architecture.

An abstract architecture defines – as any software architecture should – the basic components of the software and their interactions. Software architecture specifies an application's modularization and defines interfaces between these components. According to the Quasar reference architecture [3] software components can be classified as either technical components (T-components), which depend on the target platform, or business components (A-components), which encapsulate business functionality. An AA specifies the behavior of the business components exclusively, thereby avoiding any assertions about technical implementations.

An AA is not a reference architecture in the sense of 'one size fits all'. It is much more a tailor-made architecture which is defined only once for each multi-platform project. It is a collaborative effort among specialists of each targeted platform.

When architecting an application which is amenable to equivalence testing, the following should be considered:

- Platform independence: the AA specifies the different components and their responsibilities in an abstract way that does not contain any assumptions about or dependencies to concrete target platforms.
- Practicability: the AA must be implementable on any target platform. This implies a profound knowledge of the features and limitations of all target platforms.
- Equivalence testing framework requirements: the AA must provide a mechanism to access and replace the involved business components at runtime to enable testing.

These criteria will be discussed below in more detail.

Platform independence

Platform independence inherently prohibits any use of platform specific parts. As a consequence the AA needs to be self-contained so that it defines the complete API of all the (A- and T-) components in the system. T-components are inherently not abstract (they refer to a concrete platform). They need to be modeled in the AA by providing a platform agnostic interface. If, for instance, a graphical user interface (GUI) component is responsible for user interaction, then its interface must be part of the AA, although the graphical, haptic or acoustic representation is not part of the AA. It explicitly adapts to the target platform's user experience.

These constraints arise out of necessity: the ultimate goal of equivalence testing is to implement a single set of equivalence tests so as to be able to test all implementations. Thus tests are only allowed to access functionality which will be available on all platforms. This common functionality is determined by the AA.

Our testing infrastructure offers predefined abstract base types like integer, float and string and includes a default mapping to the obvious counterparts of the most popular mobile platforms. Every other data type must be defined in the AA. An ordered collection, such as a list, for example, needs to be modeled explicitly in the AA. Furthermore every single platform provides a corresponding type in its libraries. This is again essential to platform agnostic testing. For example, a test cannot and must not know that on one plat-

form a method is called 'length' and on another 'size'. To abstract from such naming clashes, it is common to provide a data wrapper type which implements a neutral interface and wraps the corresponding type of the target library.

Practicability

The second important requirement of a valid AA is practicability on each target platform. It must be possible to implement the specified interfaces on every selected platform without any deviations. If this is not possible, a workaround has to be found to elude the platform's limitation. Let's assume an application should be implemented for two different platforms from which one supports function overloading and the other does not. Under these circumstances the AA must not contain function overloading and would instead use distinct function names to bypass this constraint. Not all platform differences are smoothed away that easily. There are other platform specific constraints which impose much stronger restrictions on an AA.

The selection of the target platforms determines what is allowed or disallowed in an AA. To describe this more precisely we interpret the platform and its programming language as a set of capabilities. Some examples for such capabilities are inheritance, function overloading or multi-threading. The set of available capabilities which can be utilized to specify the AA is the intersection of all platform capability sets. On the other hand, all the capabilities that are not supported by all the platforms should be avoided in the AA except if there is a feasible workaround. For example, let us compare JavaScript and C# in terms of inheritance. At a first glance JavaScript does not support inheritance. However it is possible to emulate inheritance capabilities in JavaScript. If the effort to emulate inheritance on JavaScript is considered reasonable, this feature can be used in the AA. As a consequence the set of available capabilities depends heavily on target platforms. If too many capabilities are missing on a platform, its porting should then be scrutinized.

Equivalence-testing framework requirements

To be amenable to be used in our equivalence testing framework, each AA needs to offer a component registry which grants access to individual components at runtime. There are two important rules that any implementation of an AA must follow:

1. the system's components may only interact with the ones that have been retrieved from the component registry and
2. these components must communicate only through the interfaces specified by the AA.

This enables the testing framework to remotely invoke methods on components and/or replace them by mocks. A more detailed description of this is in the next section.

Scope of the abstract architecture

One reason to develop a native application is to ensure the best user experience on its target platform. GUI frameworks and user interaction concepts are usually very closely coupled to their platform. The usage of only a common (sub)set of user interface widgets that are available on all supported platforms leads to an 'unnatural' user experience or irritating GUIs. Therefore, the AA usually does not specify any user interface aspects.

Similar considerations should be applied to components that are tied closely to a platform or proprietary hardware features. It does not make sense to try to equivalence-test non-equivalent features.

The Process

The AA is the foundation for development and testing. Therefore, the interfaces have to be designed carefully. Three aspects should be considered in this context.

- The AA should not be too restrictive. A developer must be able to implement the requirement according to the given interfaces and programming conventions on his platform. For example, experience showed that developers are reluctant to adopt the AA when it imposed capitalization rules on method names (C# versus Java).
- Maintainability of code is crucial. The later a bug is recognized, the more expensive its removal will be. This general rule of software development multiplies by the number of supported platforms. Eventual changes cannot be completely avoided but an elaborated AA can minimize them.
- The components testability must be ensured. The granularity of the interfaces determines the possibilities of equivalence testing. In other words, the more details a component offers through its interfaces, the more comprehensively it can be tested.

Difficulties can also arise if a further platform is added to a project in progress. The features and restrictions of this new platform were probably not considered during the architecture phase. As a result, the existing AA might not be applicable to the new platform. Therefore, all target platforms should be ideally fixed at the beginning of the project.

A solid experience with the target platforms is essential for a good and stable architecture. Ideally each platform is represented by a skilled developer within the architecture team. During the design of the AA his task is to ensure that the common architecture complies with his platform.

If necessary, prototyping critical parts of the AA can help to discover whether they can be implemented on a specific platform. The main goal is to identify any problems as soon as possible.

Establishing an AA is a challenging task. It is about finding the right level of abstraction: being specific enough in order to build testable implementations and at the same time abstract enough to give the specific implementations freedom to make use of the peculiarities of their hosting platform.

One advantage worth mentioning about AAs is that all the development teams that work on the different platform ports have a common basis that they have to agree to. Furthermore it simplifies the exchange of ideas and experiences as it provides a common understanding and vocabulary for those involved.

Although an AA can be specified in any suitable platform-independent notation, we value UML because of the benefits of its standard representation and the diverse tools to generate code skeletons for the target platforms. Thus possible misunderstandings and misinterpretations of the AA can be avoided.

Equivalence Testing Internals

In this chapter we describe the functional principle of the equivalence testing tool along with the intentions which directed its development. The main objective is to verify that two or more implementations of an application act equivalently. This equivalence is verified with respect to a test set. Tests are written once against the AA and then applied to all specific implementations.

There are two major approaches which allow the application of one common test set to many different platforms. On the one hand, test cases could be defined in an abstract language as described in [4]. These tests then need to be translated into platform specific executable code. This system makes use of user defined transformation rules to perform this translation. On the other hand, tests can be written in a concrete programming language. The second approach requires a means for inter-platform communication to interact with test targets on different platforms.

We followed the second approach as it appears to be more promising with regard to its applicability to other projects and application domains. Our approach requires the onetime implementation of one inter- platform communication system (IPCS) per platform. This IPCS can then be reused for any equivalence testing project. In fact we provide these IPCS for Java (Android), .Net (Windows Mobile) and M-shell (an exotic Symbian based system). In comparison to the first approach, our system allows the reuse of third party libraries such as testing and mocking frameworks without additional work.

The resulting equivalence testing environment consists of a test server and test clients which connect over the network. Mobile devices equipped with the pre-installed application (the test target) plus an additional test runner software act as the test clients. The test server is responsible for handling the clients, the test execution and the test result reporting. Individual tests access transparently their mobile target through the IPCS. Entry point to the mobile target is the already mentioned component registry. This IPCS mechanism supports value and reference semantics for parameter and return values. Therefore, it is possible to pass objects between both communication partners. In the case of reference types, proxies are created on the remote site which forwards the communication to the real instance.

One big advantage of this testing infrastructure is the possibility of utilizing mocks. Mocking is an established approach to replace dependencies during component or integration testing. Mocks are configured as part of the test and the IPCS now allows these mocks to be injected into the mobile target by installing forwarding proxies into their component registry. This works, fully transparent from the perspective of both the mobile client as well as the test code. Without remote mock objects we would have to provide and maintain mocks for every platform.

Fig. 1 shows a simple example of a test execution. The goal of the test is to verify that component C1 on the test client behaves according to its specification. To test C1 in isolation, its dependency to C2 should be replaced by a mock. Let's see how this works. First the test client establishes a connection to the server. The server then locally starts a test. Inside the test, a mock object M2 is instantiated and its expected behavior is configured. To be able to invoke methods on C1 the test needs to get a handle to this component. It retrieves this handle C1' through the mandatory component registry which is exposed by the IPCS. C1' acts as a proxy for the real component C1. Now the test injects the mock M2 into the same component registry. This leads to a proxy M2' on the test client which delegates method calls to its server hosted counterpart. Finally a method on

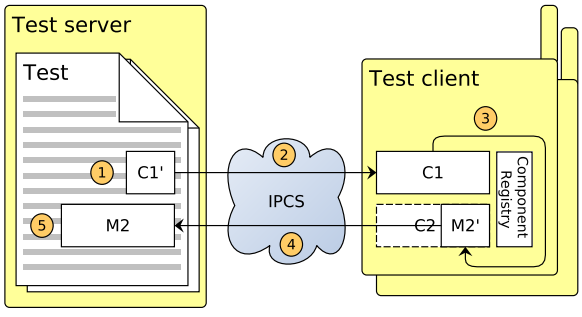

Figure 1: Overview of our testing infrastructure

C1' can be invoked (1) to verify that it returns the expected value. This call is forwarded (2) to the real component C1 on the test client. C1 requires functionality of C2 to complete the request. Since C2 is replaced by a mock, it calls a method on the injected proxy M2' (3). M2' forwards the call (4) to the mock, M2, which answers with its preconfigured return value. After all the invocations of the first call have returned (1) the test compares the returned result with the expected value and validates that the mock is being used in the intended way.

Case Study

To validate the testing approach we implemented and tested a flashcards application on two mobile platforms. In this section we demonstrate the process described in this paper on the basis of the flashcards application.

A flashcards application can manage multiple card boxes which can contain any number of cards. Card boxes are downloaded from the internet and stored on the mobile device. If the application is closed or interrupted, it must be possible to resume the learning session. The user interface should follow the platform specific guidelines.

The Abstract Architecture

Based on these requirements we defined the AA as shown in Fig. 2. The interface IFlashcardsCompReg is the component registry of the application. Starting from this interface it is possible to access the IStorageService and the IDownloadService. As the names imply the first is responsible for persisting and querying card boxes whereas the second downloads them from the web. The remaining interfaces – except IList - represent the domain model of the application.

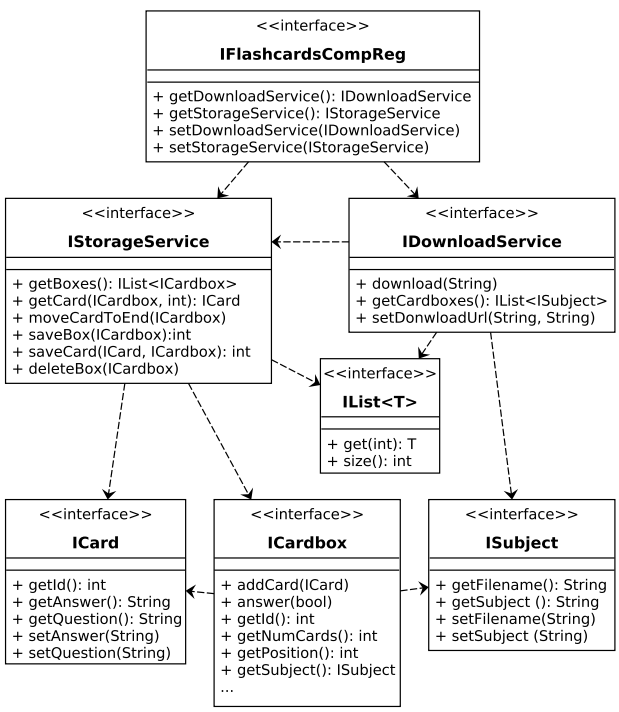

Figure 2: Abstract architecture of flashcards

This architecture satisfies the criteria described in the previous sections. It is platform independent because there are no references or assumptions to concrete target platforms. For example, the assignment of identifiers to cards and card boxes is ceded to the implementation. Another example is the IList interface. This custom interface defines a platform- independent way to work with lists. In addition a component registry makes it possible to replace real components before test runs.

All important parts of the flashcards application are covered by the AA. Tests can now verify the correctness of the download and the persistence of the card boxes and cards independent of the platform. Furthermore, the correct handling of cards during a learn session is testable (moveCardToEnd(...)). UI aspects did not affect the abstract architecture as they are handled platform specific.

Behavior Specification

So far, only the static structure of the flashcards application has been defined through the AA. What is missing is the specification of the behavior. In order to implement the AA on several platforms the developers need an accurate specification of the components, their responsibilities and how they should interact amongst each other.

We prefer test cases to specify the desired behavior. Compared to a specification document where the behavior is specified in natural language, tests have several benefits. They are executable and verify quickly if an implementation fulfills the tested behavior or not. Moreover tests positively affect software design in regards to reusability and coupling. As tests are, anyway, necessary we suggest using tests as specifications wherever possible. This also avoids the unnecessary inconvenience of having to keep the tests and the documentation in sync.

One benefit of a document in which the requirements are formulated in natural language is readability. People with a non-technical background, i.e. the customer, will be able to read such a document. Behavior driven design libraries such as specs [9] for Scala [10], fill this gap. With specs it is possible to write software specifications, which a technical layman can understand, by combining natural language with test code. The test results are presented in a comprehensible way, too. Examples will follow in the next section.

Writing Specifications

Now let us take a look at the flashcards application specifications. As our test server is based on Scala, we used specs to write specifications. In addition Mockito [11] is used for mocking. Any

```
class DownloadServiceSpec extends FlashcardsSpec {

  "DownloadService" should {
    val compReg = entryPoint[IFlashcardsCompReg]

    "invoke storage of box and cards" in {
      //init mocks
      val storage = mock[IStorageService]
      compReg.setStorageService(storage)

      //stub the mocks
      storage.saveBox(any[ICardbox]) returns 1

      //do actions
      val ds = compReg.getDownloadService()
      ds.setDownloadUrl("http://.../boxes/index", http://.../boxes/{0}")
      ds.download("Test_Card_Box")

      //verification
      there was one (storage).saveBox(any[ICardbox])
      there was atLeastOne(storage).saveCard(any[ICard], any[ICardbox])
      noMoreCallsTo(storageMock)
    }
  }
}
```

Listing 1: DownloadService specification example

other testing or mocking library running on the JVM (e.g. JUnit [12] or EasyMock [13]) would have been possible, too.

Lst. 1 shows a specification for the DownloadService. It specifies the behavior triggered by a call to the download method.

FlashcardsSpec, the base class of this specification, performs setup and teardown tasks such as cleaning up the mobile's internal storage where the card boxes and cards are stored. The StorageService methods are used for this. Besides, it provides the entryPoint method which is used to access the component registry of the application being tested. Once we have the IFlashcardsCompReg instance, we can access the defined services. The rest of the listing does not differ from a usual specification written in specs and supported by Mockito. First a mock for the StorageService is created and injected in the component registry. The mock is then programmed to return the integer 1 whenever the saveBox method is called. After that the DownloadService's download method is used to retrieve a prepared card box from a web address. If the implementation follows the defined behavior, it will connect to the webserver, download the card box and use the StorageService to persist the box and its cards. The specification ends with verifying the mock. It is expected that one card box and multiple cards have been saved.

In addition we might have checked if the card box was downloaded and parsed correctly. Special characters in the questions and answers could motivate this test. Further verifications were skipped to keep the example short. Still this specification contains all relevant parts and hopefully showed the capabilities of equivalence testing.

A comprehensive explanation of each detail and further test cases would go beyond the scope of this paper. Therefore, we would like to refer to our project site [14]. In addition the flashcards application implemented on Android and Windows Mobile is available there.

Related Work

There are different approaches to ensure that a ported application does the same as the original one. Basically these approaches are listed in [1] and can be split into three categories:
Single adaptive implementation:

- *Web applications* seem to provide an ideal solution to the porting problem but unfortunately the available browser engines interpret the HTML, CSS and JavaScript code differently. A major disadvantage is a lack of access to device features as they are shielded by browser security. Performance of JavaScript often lags behind native implementations.
- *Cross-platform frameworks* are often based on web technologies, too. Deployment is done natively on each platform. Instead of the native web browser, a web view is executing the application. This concept enables cross-platform frameworks to access device features over a JavaScript-to-Native-Bridge. A typical representative of this category is PhoneGap [7].
- *Cross-compiling* approaches translate the code of a high level programming language to native code of the target platform. Qt Mobility is a representative of this category [15].

  For single adaptive implementations, testing does not substantially differ from our approach. As tests are part of the single code base,

they can be executed on each target platform without adaption.

Deriving multiple implementations from a common model:

- A similar framework to ours is [6], where a Domain-Specific Modeling Language (DSML) is introduced to specify test scenarios. The DSML abstracts from the variability between platforms and devices. In combination with a test bed, this approach is able to automatically black box test mobile applications. A test specification interacts with the application on the same level a user does. Therefore, it is not possible to verify the system on the component level.
- Equivalence testing also follows a model driven approach: The AA represents the model against which tests are implemented. It contains the interface definitions which are used for generating proxies for mock objects.
  A pure MDA approach would supersede testing as there is only generated code. In practice only code templates are generated. These have to be extended manually with detailed code. Production and test code has to be adapted for each platform.

Manually porting an application:

- Basically this is what our approach supports: manually porting the application and then use equivalence testing to do integration tests and ensure equivalent functionality on all platforms. The "overhead" of defining an AA is amortized by the reduced effort that has to be put into testing.

Conclusion

Equivalence testing is not suitable for each kind of application. Applications that are merely GUIs for web services do not profit from equivalence testing. Equivalence testing pays off for applications with a high amount of A-components (i.e. application logic).

Seen from a testing perspective, the single adaptive implementation approach is closest to our approach. It only needs one test suite that can be run on all target platforms. Unfortunately this approach does not suit the need of all mobile development projects. This is the case if runtime performance is an issue, if the application uses features that are not supported by the cross-platform framework or if it shall adhere closely to the native look and feel.

We encourage native application development and provide a process as well as tools to pragmatically support multi-platform development. In order to have implementations with equal behavior, our tools enable component and integration testing with a single test set. All of the components that are defined in the common abstract architecture can be tested. Compared to other approaches the effort of integration test development remains constant and does not grow with each target platform.

Finally we would like to explicitly point out that equivalence testing does not replace other tests that usually are conducted during a software project. GUI or unit tests still need to be provided per platform. Equivalence testing as presented here can only be applied to components, but not to classes within these components. However, platform independent tests are written and maintained in only one place avoiding porting errors and consuming less time.

References

[1] Rajapakse D. C., 2008, Techniques for De-fragmenting Mobile Applications: a Taxonomy. Proceedings of 20th International Conference on Software Engineering and Knowledge Engineering (SEKE '08).San Francisco, USA, pp. 923-928

[2] Allen S. et al, 2010. Pro Smartphone Cross-Platform Development: iPhone, Blackberry, Windows Mobile, and Android Development and Distribution. Apress, New York, USA

[3] Haft M. et al, 2005, The Architect's Dilemma – Will Reference Architectures Help? Quality of Software Architectures and Software Quality (QoSA-SOQUA 2005), Lecture Notes in Computer Science 3712. Erfurt, Germany, pp. 106-122

[4] C. Liu, 2000, Platform-independent and tool-neutral test descriptions for automated software testing. Proceedings of the 22nd international conference on Software engineering (ICSE '00), Limerick, Ireland, pp. 713-715

[5] Miravet P. et al, 2009 DIMAG: A Framework for Automatic Generation of Mobile Applications for Multiple Platforms. Proceedings of the 6th International Conference on Mobile Technology, Application & Systems (Mobility '09), Nice, France, pp. 23:1-23:8

[6] Ridene Y. and Barbier F., 2011, A model-driven approach for automating mobile applications testing. Proc. of the 5th European Conference on Software Architecture: Companion Volume (ECSA '11), Essen, Germany, pp. 9:1-9:7

[7] PhoneGap site, http://www.phonegap.com (28. June 2012)

[8] Sencha Touch site, http://www.sencha.com/products/touch (28. June 2012)

[9] Specs site, http://code.google.com/p/specs (28. June 2012)

[10] Scala site, http://www.scala-lang.org (28. June 2012)

[11] Mockito site, http://code.google.com/p/mockito (28. June 2012)

[12] JUnit site, http://www.junit.org (28. June 2012)

[13] EasyMock site, http://easymock.org (28. June 2012)

[14] Eq-Testing project site, http://www.assembla.com/space/eq-testing (28. June 2012)

[15] Qt-Mobility a sub-project of Qt. Site: http://qt-project.org/ (28. June 2012)