Functional testing of an Android application

Funktionell testning av en Androidapplikation

Sebastian Bångerius
Felix Fröberg

Supervisor: Simin Nadjm-Tehrani
Examiner: Nahid Shahmehri
Abstract

Testing is an important step in the software development process in order to increase the reliability of the software. There are a number of different methods available to test software that use different approaches to find errors, all with different requirements and possible results. In this thesis we have performed a series of tests on our own mobile application developed for the Android platform. The thesis starts with a theory section in which most of the important terms for software testing are described. Afterwards our own application and test cases are presented. The results of our tests along with our experiences are reviewed and compared to existing studies and literature in the field of testing. The test cases have helped us find a number of faults in our source code that we had not found before. We have discovered that automated testing for Android is a field where there are a lot of good tools, although these are not often used in practice. We believe the app development process could be improved greatly by regularly putting the software through automated testing systems.
Acknowledgments

We want to thank our fellow course mates in the parallel project during the sixth semester of our program. Chi Vong, Albin Odervall, Philip Montalvo, Anton Tengroth and Jacob Nilsson were excellent to work with and together we created the application we are testing in this thesis. We also want to thank Fabian Johannsen and Mattias Hellsing for their opposition on this thesis. Lastly, we would like to thank our supervisor Simin Nadjm-Tehrani as well as Mikael Asplund for the feedback that we have received during the work on this project.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>ii</td>
</tr>
<tr>
<td>Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>Listings</td>
<td>0</td>
</tr>
</tbody>
</table>

## 1 Introduction
1.1 Why automate testing? .......................... 1  
1.2 Aim ........................................... 2  
1.3 Problem statement ............................... 2  
1.4 Approach ...................................... 2  
1.5 Delimitations .................................. 2

## 2 Theory
2.1 Testing methods ............................... 3  
2.2 Android in general and Android testing frameworks ........................................... 7  
2.3 Related Research ................................ 8

## 3 Application and testing method
3.1 Application ..................................... 9  
3.2 Testing approach ................................ 10

## 4 Test results
4.1 Map testing .................................... 13  
4.2 Contacts testing ............................... 14  
4.3 Login testing ................................... 15  
4.4 Stress testing .................................. 15

## 5 Discussion
5.1 Test results .................................. 16  
5.2 Our work in relation with existing studies ........................................... 17  
5.3 Method review ................................ 18

## 6 Conclusion .................................. 20

## Bibliography .................................. 22

## A Espresso Test Source Code .................. 24
List of Figures

2.1 General overview of a testing process .............................................. 3
3.1 Screenshots of the map and contacts activity of the application ............... 9
3.2 Flowchart of the map test ............................................................. 11
3.3 Flowchart of the contacts test ....................................................... 12
3.4 Flowchart of the login test ............................................................ 12
List of Tables

<table>
<thead>
<tr>
<th>2.1</th>
<th>Myers’ Principles of testing</th>
<th>6</th>
</tr>
</thead>
</table>
Listings

2.1 Espresso code example .................................................. 7
4.1 The space bug ............................................................ 15
4.2 The name bug ............................................................. 15
1 Introduction

As we continue deeper into the "Internet age" we become more and more connected every day. New hardware like smart phones, tablets and smart watches have already popped up, and it is hard to know what the future has to offer when it comes to computational devices for both personal and professional use. On these devices there is a lot of software for communication, shopping, etc. and more are created and updated daily. According to International Data Corporation the number of mobile applications was estimated to increase by 1600% between 2010 and 2015 \[8\] and the Google Play application market currently holds over one million applications\[1\]. These systems are becoming a larger and larger part of our lives, and as we depend more on them it is utterly important that they do still work the way they are intended to. There are a lot of aspects that define the dependability of a system, including but not limited to software, hardware and cognitive usability. In this thesis we will focus on the testing of the first subject, ignoring the latter two. Today manual testing is more widely used, even though automated solutions achieve better coverage statistics [10]. We clearly see the need for understanding how to use automated testing and what benefits can come from doing so. For this thesis we will test some functionalities of an Android application using automated testing tools publicly available.

1.1 Why automate testing?

When publishing a new application or making changes to an existing one, it is desirable to know that the application does indeed (still) work the way it is intended to. Since the number of possible inputs to the application quickly becomes unmanageable, it would be more or less impossible to test all combinations. Therefore, it is important that there exists a set of methods to test the functionality of the application without having to test all possible combinations of input. To achieve reasonable input coverage, manual tests are far from optimal. For the purpose of scalability, a methodology of automation is suitable. A good idea might be to use an existing framework that automatically finds views and tests different inputs to a computer program [9].

1.2 Aim

The purpose of this thesis is to give the reader a better view of what test automation means. This thesis will give the reader an introduction to software testing in general and then focus on software testing for Android. We will study some common misconceptions and problems faced by Android developers and developers in general. After this, we will show how this knowledge can help software developers and testers to increase the efficiency of the testing. After reading this thesis one should be able to start software testing of an Android application in an efficient manner.

1.3 Problem statement

The questions that we want to answer during this project are:

1. How can we use functional testing to find and remove errors and faults in our mobile application?
2. What are some problems that software testers face, and how can we avoid them when testing our own application?

1.4 Approach

In order to give an answer the questions in the problem statement, we chose three functions of our application to put under functional testing. We have also performed a stress test of the entire application, with the goal of finding errors that the functional tests did not find. To perform the functional tests we have used Google’s framework Espresso, while the stress tests have been performed using the Google program Monkey. We have also looked at a number of existing papers to try to relate our findings to existing scientific literature.

1.5 Delimitations

As the time and resources for this project are quite limited, we have chosen to focus our work on the Android platform, though some of the concepts mentioned will be valid for other platforms as well. Since some of the important functions of the application in the parallel group project were finished late during the work on this thesis the tests will not be as extensive as they could have been otherwise. This will not be a problem since the main purpose of the tests are to show the applicability of relevant concepts discovered through literature study.
Before introducing the testing part of this thesis, one must have some basic knowledge in the field of Android software testing. Therefore, this section is dedicated to presenting some basic background information that will be needed to understand the rest of this report.

2.1 Testing methods

The main reason to start software testing is to find faults that can be fixed to increase the reliability of the system under test (SUT) [3]. The faults might cause errors that lead to failures [2]. In this thesis the errors are divided into two categories. The first category, nonfunctional errors, consists of events such as memory leaks or runtime errors etc. These are events where the software does not work and might lead to a failure such as a crash. The second category, functional errors, has more to do with the expected behavior of the app; the software runs, but not the way it is intended to work. An example of a functional error is that images, buttons or other graphic elements are not displayed correctly, either at different times or on different devices. Another one might be that a text piece is displayed in the incorrect setting or to specific users that should not see the text [3].

Figure 2.1 represents a generalized flow of a testing process. The core of the testing framework is the “Test System” block placed in the middle. This consists mainly of an input generator. This input generator can find nonfunctional errors such as crashes but it can not find...
functional errors described above, as it will have to be provided with information about the expected output to do so. Going back to the previous example of the two categories of errors, an input generator cannot detect the error of incorrect or invisible buttons, as it has to be fed the information that this is not the expected behaviour of the system. However, such a system might be able to detect unexpected crashes and memory leaks. To extend the framework to detect functional errors, we need a specification: a set of consequences that are expected after a certain input. These are represented by the ellipse to the left in the figure. To the right side in the figure we see the "system under test" block. This is cloned into several blocks because in a lot of cases there is communication between devices and systems so it might be important to generate input to many of these instances of the SUT in the same test run, to verify that the inter-device communication is working. The last part of the flowchart is the Test outcome represented by the ellipse at the top. This is basically just a "true/false" variable, as the test system itself is fed with the specification and can therefore decide if the SUT behaves as expected or not. It would also be possible to have a scale instead of just a true/false, so that we easily can find out how bad the errors are and not just if the SUT failed to match the specification.

Test oracles

As we briefly mentioned in the start of this section, there is a difference between functional and nonfunctional errors. Barr explains this in the 2015 survey *The oracle problem in software testing* by addressing the issue of not having a good enough system available on the market to automatically detect all functional errors without a proper formal specification. The term test oracle was first used in 1978 by William Howden and refers to the part of the framework that determines if the detected behaviour is expected or not. Our categorization responds to how these oracles are classified.

The oracle decides whether the registered behavior is passing or failing the test. There are implicit and specified test oracles. An implicit test oracle can find nonfunctional errors like overflowing buffers, null-pointers, memory leaks etc. In other words it only detects the behaviour one never wants. A specified test oracle will need a specification, the expected output for each input, from every state of the application. Because of the work needed to specify all these expectations, this is far from complete automation. Even so, research of software testing frameworks has for the last thirty years been centered around these test oracles.

There have been successful attempts to generate automatic test oracles that can detect some functional errors in mobile applications. The contexts of these errors are user interaction events like "if the device is rotated from vertical to horizontal and then back, the user interface should be displayed exactly as before". This is of course not true for all applications, but for most of them.

White, grey and black box testing

Software testing methods can be divided into three categories when it comes to knowledge of the system: White, black and grey box testing. The white box testing method will test the internal workings of the application, and requires a lot of work as well as access to the source code of the application. Black box testing will test the fundamental functionality of the application, and it will not require the source code to work. Black box testing works by gathering a representation of the UI and then entering/exiting found activities as well as clicking buttons, typing into fields etc. Grey box testing is a combination of black and white box testing. It requires some insight in the structure of the code (a model of the application structure, e.g. using a Unified Modeling Language (UML) diagram) but access to the source code is not required. There are several advantages/disadvantages with these different methods, some of them are listed below.
2.1. Testing methods

- **White Box testing**
  - **Advantages:**
    The white box testing method can be used to test algorithms, as it can be fed with a way to gather/generate expected output. It can test functionality that is currently not available for the Black Box system to access, but might be exposed later.
  - **Disadvantages:**
    White box testing is often demanding and time consuming. It also has to be carried out by someone with way more expertise than needed in Black box testing. Another disadvantage is the paradox of needing a tester that has good insight into the source code, whilst still wanting someone that has not developed the system, as this person probably is less likely to find faults not previously thought of.

- **Black Box testing**
  - **Advantages:**
    The black box testing is quick and can even be fairly easy to configure. Available frameworks can often work almost on their own, for instance the android Monkey, which can be launched with only one command from the ADB command line without further configuration. It also interacts with the application in a similar way to what a user would do, and in the end that is often why we are testing an application, to make it work as expected in situations of user interaction.
  - **Disadvantages:**
    Black box can not check algorithm outputs and expected behaviour, as it works only by testing a lot of input, like a user would do. Because of this there might be bugs left undiscovered after the test has been completed successfully.

- **Grey Box testing**
  - **Advantages:**
    Compared to the black box, the grey box offers ways to test some expected behaviours, a desirable feature of a testing system. Unlike white box testing it does not require very much insight in the system, a list of activities and their respective views is sufficient. Due to this the test can be carried out by someone far less competent than what’s required for a white box test.
  - **Disadvantages:**
    As grey box tests are more loosely defined than the white and black counterparts, it is hard to determine its weaknesses. One might be that it requires some work and some insight in the application before it is possible to run the test.

**Functional testing and stress testing**

We have performed two types of testing — functional and stress testing, with main focus on the functional testing. A functional test is a form of black-box test where one, given a specific input, examines the output of the SUT to make sure that it matches the expected result. With stress testing the goal is to put the SUT under a large amount of input, often random or pseudo-random. The stress test will not find functional errors but might encounter crashes and other violations of non-functional requirements like memory leaks.

**Exploration methods**

Different testing frameworks use different exploration methods. These are the strategies used to test the various states of the Android application. Choudhary categorizes these as random methods, Model-based methods and Systematic methods. The random methods generate inputs by randomly selecting from a set of possible inputs. Some tools are limited to only
UI-events, while others are extended to enable system-events as input as well. The Model-based methods often work in a way similar to that of a Web crawler. The exploration strategy is to dynamically build a set of possible states by crawling the application’s activities. Events make up the transitions between states and when all events from all known states have been performed the model is complete, and the test is finished. Systematic methods is a collection of other ways to map the application and generate input from that.

Software testing principles

Glenford J. Myers writes in his book *The art of software testing* [12] about ten principles of software testing. These are supposed to be used as general guidelines, relevant to anyone wanting to find errors and faults of an application by using software testing. The 10 principles of testing are shown in table 2.1.

<table>
<thead>
<tr>
<th>Principle number</th>
<th>Principle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A necessary part of a test case is a definition of the expected output or result.</td>
</tr>
<tr>
<td>2</td>
<td>A programmer should avoid attempting to test his or her own program.</td>
</tr>
<tr>
<td>3</td>
<td>A programming organization should not test its own programs.</td>
</tr>
<tr>
<td>4</td>
<td>Thoroughly inspect the results of each test.</td>
</tr>
<tr>
<td>5</td>
<td>Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.</td>
</tr>
<tr>
<td>6</td>
<td>Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.</td>
</tr>
<tr>
<td>7</td>
<td>Avoid throwaway test cases unless the program is truly a throwaway program.</td>
</tr>
<tr>
<td>8</td>
<td>Do not plan a testing effort under the tacit assumption that no errors will be found.</td>
</tr>
<tr>
<td>9</td>
<td>The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.</td>
</tr>
<tr>
<td>10</td>
<td>Testing is an extremely creative and intellectually challenging task.</td>
</tr>
</tbody>
</table>

Software testing problems

In a paper by Daka and Fraser[5], the authors present the results of a survey on testing practices among software testers. The work was done by sending a questionnaire to some software developers to find out how they use unit testing to find errors and faults. Three things that they found are:

- The two main reasons that developers write unit tests are own conviction and management requirements. Customer demand is also seen as one important factor.
- Out of the total time spent in the development process, 33% is spent writing new code while close to 16% is spend writing tests. Another 25% is spent on debugging and fixing existing code.
- The single most difficult part about testing is to identify which code to test, followed by isolating the unit under test and determining what to check.
2.2 Android in general and Android testing frameworks

In this section we will describe some key characteristics of the Android operating system. We will also describe the two Android-specific testing frameworks Espresso and Monkey.

Android platform

The Android platform is widely used in mobile phones (it actually has the largest share of the mobile market [4]), tablets and even smart watches. Applications developed can be run in debugging mode on either an external Android device, or in any of the many available virtual machines.

Android has a custom Linux kernel in the bottom of its software stack. This kernel provides the main functionality of the system. Android applications are usually written in Java, although some options (e.g. C++) exist. The graphical layout can be described programmatically in Java code, but the more common way is to use the descriptive XML format instead.

Android uses identification codes (in the future referred to as id codes) to identify different views (buttons, text fields, images etc.). These id codes are what developers use to find a specific view, and is also what Espresso is able to use (explained below) to find and interact with views to perform the tests.

The Android platform is open source, which makes it easier to create testing frameworks for applications running on Android, as access to the source code is trivial[7]. Possibly due to this, many tools have been developed.

One drawback that Android has in the context of testing is the fact that it runs on so many different devices. While some companies produce their operating system and hardware together so the number of different devices are limited, Android on the other hand has to function properly on different processors, graphics units, screen resolutions, networking cards and other third-party peripherals. This makes manual tests very expensive, as the tester needs access to a large number of devices to test on.

Espresso

Espresso[1] is a UI testing framework for functional testing of Android applications, developed and maintained by Google. Since it is developed by the same company as the Android operating system it has a number of advantages compared to third-party solutions, with the main advantage being its tight integration with the OS[13].

Each Espresso command consists of up to three parts: A ViewMatcher, a ViewAction and a ViewAssertion. The ViewMatcher is used to find a view, either by using the id, the text on the view or other identifiers. ViewAction is used to perform actions on the view such as a click or text typing while ViewAssertion is used to check that a view is displayed (or not displayed) correctly [13].

Listing 2.1: Espresso code example

```java
onView(withId(R.id.myButton)).perform(click());
onView(withText("This is a text")).check(matches(isDisplayed()));
onView(withHint("Type text here")).perform(typeText("Hello World!"));
```

In listing 2.1 an example of some Espresso code is shown. In this example we use the ViewMatchers "withId", "withText" and "withHint", the ViewActions "click" and "typeText" and the ViewAssertion "isDisplayed". The test will click a button by finding its id (myButton), check if a text view with the text "This is a text" is visible and then type the text string "Hello World!" into a text field that currently holds the text "Type text here". We can see that it is a grey box test because it is a functional test, but in the first row we actually find a view by

using its ID, which is not visible to an end user of the application, thus we get into the internal workings of the application and the test is not purely black box.

**Monkey**

In the Android IDE *Android Studio* the input generation tool (exploration method) Monkey is included. It uses a random exploration method, meaning it will randomly chose actions from a set until it has reached an initially provided amount of actions to execute [4]. It is also the most popular input generator for Android [10].

### 2.3 Related Research

Android software testing is a research field that has been studied a lot by many different computer scientists. In some of the papers we have studied the authors present their newly developed framework for Android testing and present some of the advantages with their framework. One example is a paper written by Amalfitano et al. [1] where the authors present their tool called MobiGUITAR. What we have found in the similar papers is that the authors often try to promote their framework. One thing that most of the studied literature lacks is a complete comparison between some of the available testing methods. We have found a recent and quite extensive survey, comparing different input generation tools [4], but these are just for input generation and GUI crawling, not any white box testing.
In this section, the method used is described. The method mainly consists of a series of functional tests as well as a non-functional test on our Android application. Since most of the functions of the application will need to be semi-finished (functional enough to test) before starting the testing, this chapter starts with a brief description of the application under test. Several flowcharts with interactions representing those that a user would perform when using the application are presented. These flowcharts are the basis for the functional tests.

3.1 Application

Figure 3.1: Screenshots of the map and contacts activity of the application
The application that have been the subject of functional testing in this thesis is an application that we have developed in parallel with the work on this thesis. The application is a client and server proof of concept that is intended to help emergency agencies during crisis scenarios. The system uses a server written in Java and several Android clients, also running Java (in the Android format described in chapter 2). Some of the functions in the application are:

- A map view in which the user is able to view, add and share information about points of interest (POIs) with other users of the application.
- A contact catalogue which allows the user to share contact information with other users of the application.
- NFC functionality, giving the user the possibility to use the Android phone as an access card to open doors or gates with corresponding technology installed.
- A communication view which sets up a secure voice and video stream between users of the application.

Screenshots of some of the activities in the application can be seen in Figure 3.1, with the map view to the left and the contacts view to the right. The blue pins on the map view are the points of interest [POIs] placed by users of the application. These can be pressed to open more information about the specific POI. The red pin is a locally stored POI ready to be synchronized with the server.

### 3.2 Testing approach

The testing framework Espresso has been installed and tested in order to understand the basic workings. The reason we have chosen to work with Espresso is because from our point of view it is the framework that needs the least amount of configuration to work, which is also backed by a 2014 paper by Knych and Baliga [9]. In a paper by Morgado and Oaiva [11] the authors present their test generation tool iMPAcT, and also mention that Espresso is commonly used by researchers as a test case execution system which indicates that this is indeed a good choice for our testing. Another reason is that Espresso is well synchronized with the Android system. For example, the test system will wait for the UI to be in a stable state before performing the next action which means that there is no need to add a thread sleep between every UI action [9]. This simplifies the test programming and reduces the risk of test failure caused by unfinished UI rendering. The actual testing have been done by writing test cases that match the features that we want to test. The results of these tests have been reviewed and have been used to help us decrease the number of faults in our code.

None of the tests explained below require strict access to the code, but with our implementation in Espresso, they use internal Android id codes (see the first row in listing 2.1 for an example of this). Therefore the test cases can be seen as a form of grey box testing, but could easily be transformed into black box tests by using identifiers other than the id:s of each element (for example by using the displayed text instead).

We have used the Google program Monkey for the stress testing. This is primarily because it’s shown to be one of the most efficient input generators on the market [4], but also because it is very easy to set up and get to work.

### Map testing

The MapActivity has been tested using an Espresso test case. The test case works as follows: X and Y coordinates as well as a description is randomly generated. A point of interest (POI) is added to the system using the generated coordinates and descriptions. This is repeated using different coordinates and descriptions, which are also saved locally.
3.2. Testing approach

The coordinates are then pressed in the map activity, which should open another activity that displays information about the POI. The description text field is checked so that the text on the field matches the string that was previously entered. If the test fails we know that the POIs have either been placed incorrectly or saved with wrong description. The whole process is described with a flowchart in figure 3.2, where the start node is green and end nodes are red.

Contacts testing

The contacts function is tested using an Espresso test case similar to the test on the map function. A contact is created and, as with the test on the map function the text input to the application is randomly generated. The contact is uploaded to the server. After this a full server sync is performed after which the test system will make sure that the contacts that were added actually exist in the contact catalog and that the data presented is correct. This is also presented in a flowchart in figure 3.3, where the start node is green and end nodes are red.

Login testing

We have also tested the login service to the communication activity. The login function in our application works by the user typing their user name and password into corresponding fields in our settings activity. The authentication then takes place when each request is sent to the server. Every time the user sends data or a request to the server, the user name and password is submitted together with the request. The way this function has been tested is by trying various combinations of correct and incorrect user names and passwords. The test system has then entered the video call activity, as this activity is initiated with a request to log into a webRTC server. If the login is correct we are then able to see the header over the list of contacts, otherwise not. Our test is able to verify that all valid user name/password combinations are able to login, but none of the incorrect ones present in the test set. The login test flowchart is shown in figure 3.4.
3.2. Testing approach

**Stress testing**

While the Espresso tests let us isolate and effectively test a subset of the functions in the application, it may also be a good idea to try to test the entire application in one test. This has been done by using an input generator called Monkey. This has been used mainly because it is easy to set up and has proven to be a good competitor, despite its usage of a random exploration method [4]. What monkey will do is to act as a regular user, but without any intention of accomplishing anything particular. Monkey presses random views on the screen, buttons on the device and settings from the Android actions bar (visible in figure 3.1 at the top). Monkey has a set limit for the number of events to generate, and after these have been carried out the test will stop. The purpose of exposing our application to Monkey was to find errors caused by faults like unescaped characters in input fields, overflowing fields in the database when submitting data through the server and buttons that have been mapped incorrectly, etc.
In this section we will present the results of the Espresso test cases as well as the Monkey test runs explained in the previous section. During the tests we found a number of errors that were previously unknown to us and to our development group, along with some errors that we were aware of but had not yet taken time to fix. The test results along with the errors we found will be presented below. The source code for all the Espresso tests is available in the appendix.

4.1 Map testing

During our test of the map activity we found a total of 4 different errors. The errors that have been found are listed below.

1. If the server is down, or an incorrect server address has been entered in the application the app crashes instead of sending an error message to the user.

2. If a user tries to add a POI with a description containing the quotation mark (”) or other character sequences that need to be escaped (\n, \b etc.), the POI will be added to the server correctly but trying to sync the POI back to the client will cause the server thread to crash.

3. It is possible to place a POI on the map that, when returning to the full map view, is displayed behind one of the buttons in the top of the view ("Add a new POI" and "Request server sync"), that can be viewed in figure 3.1. These POIs will be inaccessible because there is no option of panning the map to view the top of it, as the map image ends there and there is no more content in the container.

4. Two (or more) POIs can be placed in the same location on the map, resulting in one of them being impossible to click. There could be a more urgent task on a POI that ends up behind a POI with a less urgent task on the same (or close to same) coordinates, resulting in an important matter not being detected.

We were previously aware of the last two errors while the first two were new to us. The first error is caused by a fault when synchronizing to the server. A socket is created for the
request, but there was no timeout when the host was unavailable so the application busy-waits for a response. This resulted in an infinite loop, which on top of it all ran on the User Interface (UI) thread causing the SUT to lock.

The second error is caused by a bug on the server side. We managed to bypass it by escaping the characters with a backslash. We traced the fault to gson[1](which is used on the server to translate the database responses to json objects that can be sent to the client). Gson will not escape the characters by itself when translating the retrieved data from the database and is therefore left with syntax errors in the json objects, as these objects are separated using quote characters (").

The third and fourth errors are somewhat different, as they have been known since before the testing. Because of the SUT being a proof of concept, the mechanisms solving these errors have not been implemented due to time constraints. One might argue that an option of panning the map past its edges would solve the third error, the fault is then on the panning feature (the scrollView containing the map). One might also argue that the third error can be solved by having an option of hiding the buttons temporarily, causing the fault to be on the outer view in the map layout. The fourth error would be solved by adding a new feature of listing all the nearby POIs when tapping an area with a high concentration of pins. The fault is therefore not a bug per se, but rather a lack of functionality or perhaps a usability issue.

### 4.2 Contacts testing

Our test on the contacts activity presented the following errors.

1. If the user adds a contact with a space in either of the fields, the space disappears when the contact is saved locally on the unit
2. If the first and last name are more than about 20 characters long combined, the name will not fit in a single line and thus causes a graphical error.
3. Using non-english characters such as ‘å’, ‘ä’ or ‘ö’ in any contact field will cause the server to give an authentication error, and the contact will not be added.

All of these errors previously unknown to us. The third error did not exist when we started testing, but was (by mistake) introduced later when trying to increase the security of the client-server communication.

The first error is caused by a bug in the contacts class. The programmer who wrote the class decided it was a good idea to use space as separator when saving the names, and thus it would not be possible to have it inside the actual name. They therefore wrote the bug shown in Listing 4.1.

The second error is caused by a bug in the XML element being inflated to hold the contact. The height of the TextView displaying the name is too small in height. It could be fixed by letting it wrap the content, rather than having a fixed height. The bug is displayed in Listing 4.2.

The third error is caused by the server having another character encoding than the client. This is therefore not caused by a syntax fault (a bug), but rather a configuration mistake. The server will replace the unknown character with a "?" and when the message’s checksum is calculated it will differ from the correct checksum where for instance the character "ä" should have been instead.

[1]https://github.com/google/gson
4.3 Login testing

When testing the login to the communication system the turnout was negative, we found no errors using our test. We ran the test with 25 iterations (test size = 25) about 3 times (three consecutive test runs) but were still not able to find any errors; the test ran to completion every time.

4.4 Stress testing

Our stress tests using the Monkey input generator resulted in only errors already discovered. We came across a combination of the first and second errors from the list in the Map testing section. Monkey would crash the server by submitting an unescaped special character and then try to contact the server, resulting in a crash on the client side. Even though we ran several Monkey stress tests of sizes ranging from 100 to 20,000 events, we did not discover any new errors.
5 Discussion

In this chapter we will analyze the results from our Espresso tests. We will also compare our first impressions with testing against what we found in the existing papers that we have studied.

5.1 Test results

Even though we only wrote three simple tests for our application we discovered a number of failures, more than we expected. Both the contact test and the map test were initially aimed at encountering problems with the user interface of the application, such as pins of the map being placed incorrectly. Instead, most of the failures were actually related to the communication between the server and the client. We discovered some connections to the fifth principle of testing\cite{12} in the form of unescaped special characters posing problems on the server side. It is not a valid input to be named \n or ”, but if typed and submitted, it should not crash the server. The reason we found this error was that we naively chose a random ASCII value amongst those representing actual characters (no mute buttons etc.) instead of typing names that would actually make sense.

Map testing

As mentioned earlier, the tests on the map function of the application gave us a somewhat surprising result. In addition to the two errors that we were aware of on the client side, the test system presented two more errors that both were related to the client-server communication. Finding these errors by manual testing would of course be possible, but since they require the person testing the application to have an understanding of the programming and having the intention of trying to crash the application, we believe that it would require extensive manual testing to find whereas a simple automated test using randomized input was able to find it quite fast.

Contacts testing

The test of the contact catalogue also gave some interesting results. Since this was probably the simplest function of the application we did not think that running a simple test on it
would find any errors. Instead, the test presented three new errors. These are all quite simple errors that we should have found during development, but we somehow managed to miss them. One thing to note about this test is that we never managed to complete the test with a pass. This indicates that there still exists bugs, either in the test code or in the program code.

Login testing

As previously mentioned we found no faults in the login functionality. We believe this to be the effect of two different things. The first is that the first part of the login test relies on very simple objects. The first part of the test is simply to input a user name and password into two Android editText text fields. This in itself is not really introducing anything new. These text fields are open source elements used in a lot of places both in our application and others, so it is no surprise that they work; they have probably been tested many times before. The second part of the login test is more complex. A request to log in is sent to the webRTC server. The credentials are the ones specified in the fields and they have to match the ones on the server when they reach it. We believe the reason the test does not discover any faults is because the server is in fact part of a library called Quickblox Our library (and especially its login functionality) has probably been tested, not necessarily with any automated solution, but by the people who have used it since its release. Therefore our fairly simple test case does not discover any weird, unexpected behaviour.

Stress testing

We believe the reason we found no new errors running the Monkey test is the simplicity of the application apart from the already tested functions. The app is a proof of concept, meaning that it only has to show the desired functions in isolated units. The software components keeping these parts together are thus very simple, not much can go wrong. Monkey will only be able to trigger a few procedures that are not just entering or exiting different activities. Another reason Monkey can’t discover too many errors is because it often enters the settings activity and changes the login credentials quite early during the run. When these credentials are incorrect, the server will not accept the requests being sent, and only a Toast with an error message will be shown.

5.2 Our work in relation with existing studies

We’ve discovered some interesting ideas and facts about software testing when studying recent literature on the topic. The principles of testing as presented by Glenford J. Myers that we introduced in section are very much related to what we’ve found in articles about software testing. We found that testing is a destructive process where the goal is to find bugs, not ensuring that the application works. Of course both the customers’ and the developers’ goal is reliable applications, and this is why the developers should not be the ones testing the application. If the developers would do the testing, they would be stuck in the paradox of trying to prove that their own code is malfunctioning. While they want to be successful in their testing (find and remove bugs), they also want to avoid showing they made mistakes to begin with. When it comes to test oracles, we found that the research for many years has been focused on specified test oracles. Ideally there would be advanced enough implicit test oracles available, that would be able to find all sorts of errors. For instance this could be achieved by letting an AI observe a great amount of testing routines and then write the next specification by itself. It could be tuned to value recall higher than precision and thus find most of the errors but also false positives. This would move the work from writing specifications to reviewing found errors. This would probably result in a much more efficient testing

process as it would reduce or even nullify the amount of work needed to write extensive specifications of expected behavior.

As for our testing, the application is a proof-of-concept and thus there is no need to investigate all the errors found in the test runs. Many of the bugs are related to functions that are not part of the required functions so technically they are not part of the project. This of course removes the paradox where testers do not want to find bugs in their own application and since we had no other option than testing our own application those principles become useless as well, but we still think that we can relate to the other principles presented by Myers\cite{Myers2004}. For example, principle 5 (test cases must be written for invalid input) is what helped us find a lot of the errors presented in chapter\cite{chapter4}. Myers\cite{Myers2004} also states in the 9th principle that “The probability of the existence of more errors in a section of a program is proportional to the number of bugs already found in that section” which is something we strongly believe in. Considering that our simple tests found a lot of errors, a full test with better test cases is likely to find even more problems.

Although we are able to relate to the principles of testing we are finding it harder to relate to all of the software testing problems found in a survey by Daka and Fraser\cite{Daka2015}. We think that the main reason for this is that the survey was sent to experienced developers and testers. Since the authors of this thesis as well as the coworkers in the application development process are new both to the development method used and to software testing it is hard to compare the time spent on code writing to the time spent on writing tests. When starting the development of the application, no one in the group suggested any form of software testing of the application. Again, this could be explained by the fact that we were all new to this form of software development and is something that probably will change in future projects. Daka and Fraser\cite{Daka2015} also found that the hardest two parts about testing is to identify which code to test and isolating the unit under test. Both of these points are things that we understand and have noticed ourselves during this project and will be discussed in section 5.3.

5.3 Method review

We review the method as a reflection on learning and to help anyone else who is in the process of testing an application or producing a thesis similar to this one. Although we have taken precautions when deciding on what method to use, we can still see some risks of drawing false conclusions. There have also been lessons learned about the execution of the tests.

Choice of tools and resources

One core choice is the choice of using Espresso as the Android testing framework in our own tests. Although we have some resources indicating that Espresso is a good choice\cite{Espresso}, we have not conducted any comparative study of our own. There might also be fallacies caused by inaccuracy in the studied literature. To counter this we have tried to gather information mainly from peer-reviewed articles. Hence, we feel more confident about our choice of Monkey since it relies on a thorough and recent study\cite{Monkey}. We have also used a book by Glenford J. Myers to learn how to test applications efficiently. Myers’ book has been published in many editions since it first came out in the 80s and is one of his most successful publications \cite{Myers2004}, which is why we chose to trust some of his ideas (at least as relatively trustworthy opinions).

Some decisions regarding the choice of method(s) have been considered by us to be more or less trivial. For instance we have used Android studio as our IDE, where there are others (like intellij or Eclipse) that would have worked as well. We have also only applied or tests on three devices, all mobile phones. The app is supposed to work on tablets as well, and should therefore also be tested on a tablet to ensure that we find all errors.
Execution

Since none of the authors were familiar with any form of testing before it seemed reasonable to start the project by studying existing papers to get an idea of some common traps that software testers fall into. Looking back at the project we still think that this was a good approach since it made it possible for us to avoid some dead ends that we otherwise might have had problems with.

Regarding the testing part of the project we think that there are some things that we could have done better. When examining the results of our own tests, it was sometimes hard to tell what caused the error in a failed test case. When looking back at our test cases we can see that all of them try to test a big chain of events, which makes it hard to locate the fault. This could be prevented by trying to separate the tests into smaller parts, which is something that we will try to do the next time.

We have also noticed that we did not test all functions of the application, some because they were hard to write proper tests on and some because they would require a more advanced testing framework with support for multiple devices to work. There is also at least one error that we are aware of that our test cases did not find. This is because it is in a part of the code that was not fully tested.

In the future we will take more time to plan the tests before starting writing them. We think that this will help improve the quality of the tests which will help us to identify more errors, fix the corresponding faults and thus increase the quality of the software even more.
Conclusion

After our tests and our studying of existing papers we can conclude that Android software testing is an interesting field with some special characteristics. When starting off our work with this thesis we believed the choice of framework would need to be firmly supported before starting to carry out tests. We can now conclude that the most important parts of the testing process is having the right mindset (a destructive one), writing many good test cases and, if applicable, use a good exploration method that efficiently explores the application structure. There have been several comparisons between testing frameworks, but most of them end up with the result of easier solutions being better. Just the discovery that a random exploration like Monkey beats strategic counterparts in efficiency is surprising [4]. For this thesis we decided to work with Espresso which seems as a reasonable choice since it fulfilled all of our requirements and is widely used in the scientific community[11].

When it comes to our own application, we can conclude that it was not as robust as we initially thought. We had thought of some issues that might occur (POIs moving around etc.) but did not expect to find so many errors related to the characters in the strings submitted to the server/database system. Our testing of the application has aided the development of it greatly. We used some of our error-finds in our application demo to exhibit the abilities and limits of the system. For example we found that the server crashed when submitting a point of interest named “", which we showed to our mentors during the demo. If we would have been more used to application testing when starting our parallel project (the development of the app) we would definitely weave it into the development process and probably benefit a lot from doing so.

For our future tests we have identified a number of things that could be improved, where the most important thing is that we should have created more but smaller tests to isolate the code under test in a better way. This is something we noticed in all test cases but it became most obvious in the testing of the contact function of the application where we were unable to find all faults causing the test to fail. Smaller tests would have helped us identify these faults. We were also unable to test all parts of the code, and increasing the coverage is something we will work more on in future tests.

The things mentioned in the previous paragraph is also what we would like to recommend all software developers to consider before initiating their first functional tests. We would also like to see some more research done in the field of smarter, more advanced implicit testing oracles. When it comes to mobile applications we can see a lot of similarities
between them: a hint that there might be a possibility of making an automatic testing framework that uses an implicit or at least semi-implicit testing oracle that efficiently can find failures[14]. We recommend developers to keep an eye out for papers presenting any systems like this, and we would also like to encourage developers to build such systems themselves. Until we have those tools though, we suggest using the tools the developer finds easiest to get to work. The benefits of using an optimal system is rather small compared to the difference between effectively writing tests and writing no tests at all.
Bibliography


Map Test

```java
public class MapTest extends ActivityInstrumentationTestCase2<MapActivity> {
    private int runs = 5;

    public MapTest() {
        super(MapActivity.class);
    }

    @Override
    public void setUp() throws Exception {
        super.setUp();
        getActivity();
    }

    public void testAddPOIs() {
        Random r = new Random();
        PointOfInterest[] poiArray = new PointOfInterest[runs];
        onView(withId(R.id.serverSync)).perform(click());

        for (int i = 0; i < runs; i++) {
            String desc = generateRandomString();
        }
    }
}
```
int xCoord = r.nextInt(1080);
int yCoord = r.nextInt(1450) + 200;
onView(withText("Add a new POI")).perform(click());
onView(withId(R.id.addPOIFromMapButton)).perform(click());
onView(withId(R.id.mapImageView)).perform(clickXY(xCoord, yCoord));
onView(withId(R.id.addPOI)).perform(click());
onView(withId(R.id.addPoiDescription)).perform(typeText(desc));
closeSoftKeyboard();
onView(withId(R.id.addPOIButton)).perform(click());

poiArray[i] = new PointOfInterest(yCoord, xCoord, desc);
}

for (int i = 0; i < runs; i++) {
onView(withId(R.id.mapImageView)).perform(clickXY((int) poiArray[i].getLatitude(), (int) poiArray[i].getLongitude()));
onView(withText("Description: " + poiArray[i].getDescription())).check(matches(isDisplayed()));
pressBack();
}

public static ViewAction clickXY(final int x, final int y) {
return new GeneralClickAction(Tap.SINGLE,
new CoordinatesProvider() {
@Override
public float[] calculateCoordinates(View view) {

final int[] screenPos = new int[2];
view.getLocationOnScreen(screenPos);

final float screenX = screenPos[0] + x;
final float screenY = screenPos[1] + y;
float[] coordinates = {screenX, screenY};

return coordinates;
},
Press.FINGER);
}

private String generateRandomString() {
Random r = new Random();

```java
int num = r.nextInt(50);
StringBuilder stringBuilder = new StringBuilder();
for (int i = 0; i < num; i++) {
    int temp = 0;
    do {
        temp = r.nextInt(94) + 32;
    } while (temp == 34);
    stringBuilder.append((char) temp);
}
return stringBuilder.toString();
}
```

### Contacts Test

```java
/*
The imports have been excluded
*/
/**
 * Created by Felix & Sebastian on 2016-04-19.
*/
public class ContactsTest extends ActivityInstrumentationTestCase2<
    ContactsActivity> {

    public ContactsTest() {
        super(ContactsActivity.class);
    }

    public void setUp() throws Exception {
        super.setUp();
        getActivity();
    }

    public void testContacts() throws Exception {
        int contactsToFill = 2;
        onView(withId(R.id.getcontacts)).perform(click());
        onView(withText("Root Toor")).check(matches(isDisplayed()))
            .perform(swapoEncrypt());
        Set<Contact> set = new HashSet<Contact>();
        for (int i = 0; i < contactsToFill; i++) {
            set.add(fillInContactForm());
        }
        swapToList();
        onView(withText("Sync unadded contacts")).perform(click());
        onView(withText("Get contacts")).perform(click());
```
for (Contact cc : setet) {
    onView(withText(cc.getFirst_name().concat(" ").concat(cc.getLast_name()))).check(matches(isDisplayed()));
}

public void swapToCreate() {
    onView(withText("Create contact")).check(matches(isDisplayed()));
    onView(withText("Create contact")).perform(click());
}

public void swapToList() {
    onView(withText("Contact list")).check(matches(isDisplayed()));
    onView(withText("Contact list")).perform(click());
}

public Contact fillInContactForm() {
    Contact c = new Contact(1337, generateRandomString(),
            generateRandomString(), generateRandomNumber(),
            generateRandomEmail());
    onView(withHint("First name")).check(matches(isDisplayed()));
    onView(withHint("First name")).perform(typeText(c.getFirst_name()));
    onView(withHint("Last name")).check(matches(isDisplayed()));
    onView(withHint("Last name")).perform(typeText(c.getLast_name()));
    onView(withHint("Phone")).check(matches(isDisplayed()));
    onView(withHint("Phone")).perform(typeText(c.getPhone()));
    onView(withHint("Email")).check(matches(isDisplayed()));
    onView(withHint("Email")).perform(typeText(c.getEmail()));
    closeSoftKeyboard();
    onView(withId(R.id.addContact)).perform(click());
    return c;
}

private String generateRandomNumber() {
    Random r = new Random();
    int num = r.nextInt(29);
    StringBuilder stringBuilder = new StringBuilder();

for (int i = 0; i < num; i++) {
    int temp = r.nextInt(9) + 48;
    stringBuilder.append((char) temp);
}
return stringBuilder.toString();
}

private String generateRandomEmail() {
    Random r = new Random();
    int num = r.nextInt(13);
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < num; i++) {
        int temp;
        do {
            temp = r.nextInt(94) + 32;
        } while (temp == 32);
        stringBuilder.append((char) temp);
    }
    stringBuilder.append("@");
    num = r.nextInt(10);
    for (int i = 0; i < num; i++) {
        int temp;
        do {
            temp = r.nextInt(94) + 32;
        } while (temp == 32);
        stringBuilder.append((char) temp);
    }
    stringBuilder.append(". ");
    String[] domains = {"com", "se", "nu", "org", "jp", "info", "tv", "it", "pl", "ch", "co.uk", "us", "me"};
    num = r.nextInt(domains.length - 1);
    stringBuilder.append(domains[num]);
    return stringBuilder.toString();
}

private String generateRandomString() {
    Random r = new Random();
    int num = r.nextInt(29);
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < num; i++) {
        int temp;
        do {
            temp = r.nextInt(94) + 32;
        } while (temp == 32);
Login Test

```java
/*
The imports have been excluded
*/
/** Created by Felix & Sebastian on 2016-05-06. */

public class LoginTest extends ActivityInstrumentationTestCase2<
    MainActivity> {
    Map<String, String> validLogins;
    Map<String, String> invalidLogins;
    public LoginTest() {
        super(MainActivity.class);
    }

    public void setUp() throws Exception{
        super.setUp();
        getActivity();
    }

    public void testLogin() throws Exception {
        Random random = new Random();
        int testSize = 25;
        validLogins = new HashMap<>();
        invalidLogins = new HashMap<>();

        validLogins.put("root@domain.com", "password");
        validLogins.put("email@domain.com", "password");
        generateInvalidIdentities(20);
        onView(withText("Settings")).perform(click());
        for (int i = 0; i < testSize; i++) {
            boolean validIdentity = false;
            String username;
            String password;
            switch (random.nextInt(2)) {
                case 0:
                    username = (String)(validLogins.keySet().toArray()
                        ()[random.nextInt(validLogins.size())]);
                    password = validLogins.get(username);
                    validIdentity = true;
```

```
break;
default:
    username=(String)(invalidLogins.keySet().toArray())[random.nextInt(invalidLogins.size())];
    password=invalidLogins.get(username);
    validIdentity=false;
    break;
}
onView(withId(R.id.identity)).perform(replaceText(username));
onView(withId(R.id.identityPassword)).perform(replaceText(password));
pressBack();
onView(withText("Communication")).perform(click());
Thread.sleep(100);
try {
onView(withText("Select contacts to Call")).check(matches(isDisplayed()));
pressBack();
} catch (Exception e){
    if (validIdentity){
        // crash test run by trying to click view that does not exist
        onView(withText("crash the application")).perform(click());
    } else { // Expected behaviour
    }
}
pressBack();
onView(withText("Settings")).perform(click());
}

private void generateInvalidIdentities(int noOfIdentities){
    Random random = new Random();
    for (int i = 0; i < noOfIdentities; i++) {
        switch (random.nextInt(5)) {
            case 0:
                invalidLogins.put(generateRandomEmail(), generateRandomString());
                break;
            case 1:
                invalidLogins.put(generateRandomString(), generateRandomString());
                break;
            case 2:
invalidLogins.put((String)(validLogins.keySet()
  .toArray())[random.nextInt(validLogins.size())],
generateRandomString());
break;

private String generateRandomEmail() {
    Random r = new Random();
    int num = r.nextInt(13);
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < num; i++) {
        int temp = 0;
        do {
            temp = r.nextInt(94) + 32;
        } while (temp == 32);
        stringBuilder.append((char) temp);
        stringBuilder.append("@");
    }
    for (int i = 0; i < num; i++) {
        int temp = 0;
        do {
            temp = r.nextInt(94) + 32;
        } while (temp == 32);
        stringBuilder.append((char) temp);
        stringBuilder.append(".");
    }
    String[] domains={"com","se","nu","org","jp","info","tv","it","pl","ch","co.uk","us","me"};
    num = r.nextInt(domains.length - 1);
```java
    stringBuilder.append(domains[num]);

    return stringBuilder.toString();

    private String generateRandomString() {
        Random r = new Random();
        int num = r.nextInt(29);
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < num; i++) {
            int temp = 0;
            do {
                temp = r.nextInt(94) + 32;
            } while (temp == 32);
            stringBuilder.append((char) temp);
        }
        return stringBuilder.toString();
    }
```
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Sebastian Bångerius
© Felix Fröberg