Test Report

Introduction

Purpose of application

To wake its user from sleep by challenging them to solve an easy task, making sure that they don’t simply turn off the alarm and go back to sleep.

General characteristics of application

Alarmed+ is a service for setting alarms that are deactivated by solving a task. The service stores the alarms on your device, no network connection is necessary to use the service. Examples of tasks available is solving a game of Memory or solving a number of math problems in a row.

The service will offer the possibility to add several alarms at different times, select different alarm tones, select the volume of an alarm, make an alarm recurring and select different tasks to deactivate the alarm with.

Test Environment

See developer manual (under Ant installation) for how to setup the testing environment. The testing code is located in a sub-project of the main project, in the AlarmedTestTest directory in the root. Ant is used to launch the tests.

If you have set up your project in eclipse and imported both the main project and the test project. You can simply run all the rests by right clicking on the project and then Run As->Android JUnit Test. As mentioned previously, you can read more about this in the Developer manual.

Hardware environment

We have been using a Jenkins continuous integration server for automated testing, this uses an emulator. When testing locally, such as before pushing a commit to github, either a physical device or an emulator may be used.

The following devices have been used:
Samsung Galaxy S3 - Android 4.1
Samsung Galaxy S2 - Android 4.0.4
HTC One X - Android 4.1

Emulator on OS X 10.8: Android 2.3.3 - API Level 10

The jenkins server is located at jenkins.pepparkakan.net.

Software environment

Our continuous integration server tests against an Android version 4.0.4 (Ice Cream Sandwich) emulator, using Apache Ant (version 1.8.2). Locally we have been doing testing against both Gingerbread and Ice Cream Sandwich emulators.

Softwares

Apache Ant (version 1.8.2)
Android Development SDK
Eclipse and the Android Development Plugin

System information

System version

Alarmed v0.3

Known bugs and limitations

- #5: Doesn't return to the previous application after the alarm is activated and then deactivated. Acceptance test #7.3

Test specification

see Appendix: Acceptance Tests

Automatic test

Code coverage

Overall:
Line coverage - 56,7%
Block coverage - 57,9%
Method coverage - 57,6%
Class coverage - 58,3%

Model:
Line coverage - 90,7%
Block coverage - 88,9%
Method coverage - 96,3%

Controller:
Line coverage - 77,0%
Block coverage - 67,1%
Method coverage - 84,2%

We use the code coverage tool called “EMMA”, which is a open source toolkit for measuring and
reporting Java code coverage.

We always try to reach 100% coverage, but in some cases it is hard to reach and therefore we have chosen 85% coverage as our target. Sometimes its simply not possible and therefore you shouldn’t spend too much effort on it.

We have chosen to focus on pure logic code when writing tests. By logic code we mean objects that are considered model objects, containing business logic for example an Alarm. The primary reason for focusing on pure business objects for unit testing is that it is the heart of our application, but also, being decoupled by design, it is quite easy code to test. With a well tested core it becomes much easier to scale to our application. After adding additional layers and complexity increases, if errors then start to arise it is a lot easier to debug if you know that your core is working as expected. Errors in the user interface are easier to find and correct than in the core of the application.

However we also unit test our controllers (not activities), though they can sometimes have necessary dependencies which make testing much harder and sometimes even impossible. Therefore we have not set a specific limit and are instead focusing on covering as much of the logic that is testable as possible. Which means that the 85% rule still applies to the testable logic.

Nightly builds

We don't exactly have "nightly builds", but when a commit is pushed to our git repository, a build is commenced by our continuous integration server, these builds are in every way comparable to a nightly build. Since these builds are tied to github, and as such - by way of commit messages - automatically have some limited changelogs, perhaps they are even better.

Unit test

We have been writing junit tests using the the built-in testing framework in the Android SDK. This means that if a controller class depends on the Android Context we can still test it using the AndroidTestCase. Since we are using a continuous integration server we are not as negatively impacted by the need of the android testing framework to actually run the tests on an emulator or actual device. One of the benefits from being able to use context in our tests is being able to run tests where we calculate the alarm time on the actual system where it is used. This increases our faith that the unit tests and code are working correctly.

Instead of writing robotium tests for our activities we have chosen to focus on acceptance tests. Testing an interface is hard and we believed that introducing robotium would only slow us down rather than help us. The biggest downfall of not using automated acceptance tests, is the automation part. However, having a simple interface, we believe that by introducing GUI tests we would only have introduced unnecessary complexity to our project.

Though we have a simple user interface for creating and managing alarms, the activities created when an alarm triggers are much too complex for automated testing. An obvious aspect is audio and vibration, but if we had automated the solving part of the modules we would have missed one very important aspect, actual human testing. Since we are instead using acceptance tests for this part of the system, alongside testing that it works, we are able to document how we feel about the problems. Take the math module for example: Is it too hard? Is the number of problems the user has to solve before turning off the alarm too large? These kinds of questions obviously cannot be answered by a computer... yet!

We are not testing the database because of how the SQLiteDatabase behaves and because of its dependency on the android system. Instead we are using a Mock-implementation to test the systems depending on the database. We chose to do it this way because of how difficult the class was to test. We depend on the acceptance tests to see if the database respond correctly to user interaction.

Test table

See “Test Report Table” for information about how the tests went.