UNIVERSITY OF WATERLOO

Software Engineering

An Evaluation of Testing Frameworks for Continuous Integration of Android Applications

Remind
San Francisco, CA, USA

Prepared by
Sahil Jain
Student ID: 20512777
User ID: s44jain
2B Software Engineering

May 11, 2015

Sahil Jain
6435 Valiant Heights
Mississauga, ON L5W 1E2

May 11, 2015

Dr. A. Morton, Director
Software Engineering
University of Waterloo
Waterloo, ON N2L 3G1

Dear Dr. A. Morton:

The enclosed report, entitled *"An Evaluation of Testing Frameworks for Continuous Integration of Android Applications"*, is my first work term report. This report was prepared as my 2B Work Report for the University of Waterloo. This report is in fulfillment of the course WKRPT 200. It is related to my work at Remind, where I was employed prior to my 2B study term.

Remind is a mobile messaging platform which provides a safe way for teachers to communicate with students and their parents. Their vision is to connect every teacher, student, and parent in the world to improve education.

I was on the Android team within the Engineering department at Remind. My manager was Jason Fischl, the VP of Engineering. I was responsible for developing their Android app, which included developing features, fixing bugs, and writing thorough tests. I was also responsible for managing the Continuous Integration system which builds and tests our app.

This report is a comparison between the Robolectric unit testing framework for Android, and the Espresso testing framework. I worked with both of these frameworks to improve the test coverage, build speed, and test suite success rate of Remind's Android client.

I wish to thank Jason Fischl, my manager and Aravindkumar Rajendiran, my mentor, for their assistance throughout my work term.

I hereby confirm that I have received no help, other than what is mentioned above, in writing this report. I also confirm that this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Sahil Jain
Student ID: 20512777

Executive Summary

Remind, one of America's fastest-growing ed-tech start-ups, developed an Android client in 2014 to allow teachers, students, and parents to communicate. At the time, there were only a few developers on the Android team, and the app was in its prototypical stage. As the Remind Android app became more sophisticated, and the number of contributors to the app grew, a host of issues regarding automated testing began to rise. Our Continuous Integration system would queue up, because pull requests were being opened more frequently, and the duration of each build increased due to increase in number of tests. These issues were making it difficult to scale the team size to accommodate more contributors, to open frequent pull requests, as well as to trust the reliability of tests.

This report provides a comparison of two Android automated testing frameworks: Espresso, which sits on top of the Android Instrumentation testing framework, and a newer Android unit testing framework developed by Pivotal Labs, Robolectric. This report evaluates the alternatives against a set of 4 criteria: reliability of the test suite, speed of test execution, code coverability and test code complexity. This report capitalizes Multi-Criteria Decision Analysis (MCDA) in order to provide an objective and qualitative analysis of the alternatives. [1]

Both Android testing frameworks perform on a similar level in terms of code complexity. However, in terms of test execution speed, and reliability of the test suite, the two alternatives behave vary differently. Since Robolectric mocks out the Android OS into shadow components, many operations which would normally take hundreds of milliseconds now complete instantly. This significantly improves test execution speed. Furthermore, Robolectric runs in a Java Virtual Machine instead of in an Android emulator, so the test suite becomes perfectly reliable.

Ultimately, the analysis indicates that the Robolectric testing framework is much better for testing Android applications in a Continuous Integration system than Espresso. In terms of test suite reliability and speed of test execution, Robolectric performs significantly better. This report highly recommends that any team developing an Android app with a Continuous Integration system write tests using the Robolectric framework.

Table of Contents

List of Figures

List of Tables

1 Introduction

The aim of the Android team at Remind is not only to have complete code coverage by tests, but also to be confident that if every test is successful, then the app is completely functional and bug free. Practically, complete code coverage and full confidence cannot be achieved by any testing suite, due to human error while writing the tests, as well as random error when the app is executing on actual user devices. Human error, more specifically, can be caused by not covering every edge case of a function's possible input, or not testing every possible scenario of a product feature. Random error includes the fact that there are so many Android OS versions and physical Android device models, as well as the fact that each device will have different processes running while the Remind app is running, that it is impossible to test the app in all of these environments in an automated fashion. The amount of scenarios that are testable can be greatly increased if different types of tests are written. Remind has currently taken the approach of writing both unit tests and functional tests.

Currently, all automated tests for the Remind Android client application are written using the Espresso framework, which wraps the Android Instrumentation testing framework. When any Pull Request is opened to the Remind Android source code repository, the full suite of automated tests will run against the Pull Request merged with the latest version of the main codebase. This testing is run using a Continuous Integration (CI) system, which monitors our GitHub repository for new Pull Requests. Continuous Integration is a software development practice where each developer checks in their code to a shared repository several times a day. The Continuous Integration system the Remind Android team is using is Jenkins CI. This CI system needs a physical computer to run tests on as a slave, hence it uses a Mac sitting in-house as a build slave.

The Remind Android team is currently facing a few issues regarding how automated tests are being run in the Continuous Integration System. These issues range from not being able to run tests in the cloud, to issues regarding the reliability of test results, to issues regarding the speed and performance of the test suite execution.

This report compares the Espresso and Robolectric testing frameworks for Android applications. More specifically, this report analyzes which testing framework is optimal in a Continuous Integration environment. This report first provides some background on the Pull Request Workflow, and some additional information about automated testing. Next, it defines a set of design constraints and evaluation criteria, and provides an overview of the alternative solutions. The report then evaluates each alternative against each criterion.

The analysis will include performance numbers involving speed and reliability, code coverage and code complexity concerns. These findings were determined during the investigation and subsequent implementation of unit tests using Robolectric. The performance comparisons are between two similar test suites, one written for Robolectric, and the other written for Espresso. Each test suite has 50 tests in a total of 10 categories. Finally, the report summarizes the results of the analysis, draws conclusions, and makes a recommendation.

This report is intended for those who wish to write a suite of automated tests for an Android application. The reader should have a background in software development and unit testing. It will be helpful for the reader to know Java, and to know how to develop applications on the Android platform.

2 Background

2.1 Pull Request Workflow

When a developer wants to contribute source code to the Android application repository on GitHub, they must package their changes into a Pull Request. A Pull Request is a proposed list of changes to the project's source code. When this Pull Request is opened, the full suite of automated tests will run on the Continuous Integration system. The Pull Request will then be marked as green (when all the tests run successfully), or it will be marked as red (one or more tests failed). Once the Pull Request is marked as green, a developer will review the source code changes and approve the Pull Request for merging, or reject it with comments for justification. Once the Pull Request is reviewed, the creator of the Pull Request will merge the changes to the main codebase. See figure 2-1 for a visualization of this Pull Request workflow.

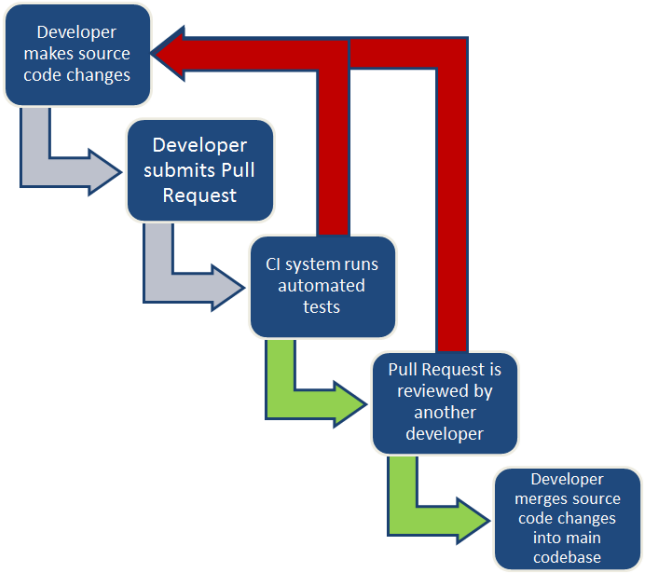

Figure 2-1: The Pull Request Workflow Diagram

2.2 Automated Testing

An automated test is a procedure which executes a set of related functions in production code, and then asserts properties about the state of the application. More specifically, unit tests assert properties about a specific programmatical function, and they stub everything that is not relevant to the function that is being tested. Unit tests are very isolated, and

each test should only cover a very small segment in code. Functional tests on the other hand, are high-level tests to assert the completeness and correctness of a feature. Functional tests usually test a large set of programmatical functions at a time. [2]

2.3 The Importance of Tests

Having automated tests of production code allows for code flexibility, maintainability, and reusability. This is because tests allow us to change our code without fear. Without tests, it is possible to introduce a bug with every change. Tests prevent regression, and tests allow the developers to refactor code and improve the architecture of code without fear that they will be breaking key functionality. Tests can be run on every Pull Request, and can act as an immediate guard before any changes are made to the production code. [3]

2.4 Espresso Testing Framework

Espresso is a framework which extends the Android Instrumentation testing framework. Test cases written in Espresso must be run on an Android device, or an Android emulator. Espresso makes it easier to perform UI operations such as clicking and dragging, and makes it easy to check the conditions of a screen (for example, if an element is visible). Espresso excels at functional testing, but unit tests can also be written. [2]

2.5 Robolectric Testing Framework

The Robolectric testing framework is unique in that it does not require an Android device or emulator to run tests because it does not test the packaged app. Robolectric will not compile your app into a `.apk` file (which is a composite of `.dex` files). Rather, the framework will compile your Java code into `.class` files, and deploy them on a Java Virtual Machine (JVM). Because Robolectric can run the test suite directly in Java, rather than in the Android OS, the tests can run significantly faster. Because Robolectric mocks out the Android OS, rather than running on top of it, the reliability of tests becomes 100%. For example, when the Android app wants to save data to the local database on the phone, it will perform an asynchronous function to accomplish this. With Robolectric, this saving of data will not happen, the data will simply be retained in memory. This means that the

asynchronous function is never called, and a race condition will never occur. Robolectric excels at unit testing, but with a bit of boilerplate code, functional tests can also be written.

2.6 Current Issues

The current Android testing framework being used at Remind is Espresso, and it has experienced a number of issues. These problems range from running tests in the cloud, to reliability issues, to performance issues.

Firstly, we cannot use any CI systems in the cloud – such as Travis CI or Circle CI. This is because Espresso tests require an Android emulator which has visible elements, but cloud CI systems can only provide headless environments to run the emulator. It would be beneficial to have CI systems running in the cloud so we can have concurrent builds, and we don't have to maintain and upgrade a physical machine in the office.

Secondly, we are having reliability issues regarding false test results. Since Espresso tests primarily operate by performing screen clicks and drags on a timer on the Android device, the tests will fail if the Android OS is too slow or fast to perform an operation. This often results in false negative test results, which break the integrity of our test suite overall. We need a test framework that will not give us false negative test results as often as Espresso does.

Lastly, we are having performance issues regarding the speed it takes to run tests. It currently takes too long to run our test suite. Our Continuous Integration system currently places all Pull Request builds in a single queue. This means that if Pull Requests are submitted faster than their test suites are executed, the queue will increase in depth. We need our tests to execute faster to prevent our pull request builds to be placed in a long queue. This fast feedback is necessary so the developer can fix potentially broken functionality quickly.

3 Analysis

3.1 Design Constraints

The Android testing framework we end up choosing as our accepted solution must fulfill a number of requirements:

- The framework must be able to operate on Macintosh OS, which is used for development by the Android Engineers at Remind
- The framework must be able to operate on Linux OS, which is part of the environment where the tests will be run on cloud-based CI systems such as Circle CI
- The test suite must be executable by running a single script or command in the terminal; it should be a trivial operation for the developer to run tests in a local environment

The Android testing frameworks evaluated by this report adhere to these technical constraints.

3.2 Evaluation Criteria

To decide which testing framework, Espresso or Robolectric, is most suitable for automated testing of the Remind Android app in a Continuous Integration environment, a number of evaluation criteria were used. On the quantitative side, the speed of test execution is compared. Ideally, we want the test suite to run in 3 minutes or less. Furthermore, reliability of test results is compared between the solutions. The test suite must provide false test results as rarely as possible. Ideally, the test suite will provide false test results less than 5% of the time.

On the qualitative side, the code complexity of test cases will be compared. The accepted solution must allow developers to write concise test cases, without too much boilerplate code or convoluted algorithms. The difference in test code complexity between the accepted solution and the current implementation should be trivial. The code coverage of each testing suite, Robolectric and Espresso, will also be compared. The accepted solution should be able to cover as much production code as possible.

The quantitative criteria will be weighed significantly more than the qualitative criteria. This is because the performance and reliability requirements are more important than code complexity or code coverability.

3.3 Criteria Weighting

Multi-Criteria Decision Analysis (MCDA) will be used to calculate the overall accepted solution for this report. [1] MCDA requires that each criterion be weighted according to its importance in deciding which alternative solution to accept. A simple justification of the criteria weightings is within the scope of this report, and is shown in table 3-1. Reliability of the test results is by far the most important. This is because a false test result, whether it is a false positive or false negative, has an extreme negative impact on the reliability of the Continuous Integration system. The purpose of the CI system is to allow or block code from being merged into a central repository, so if the CI system is blocking code that works, or passing code that doesn't work, the quality of the production code is directly degraded. Speed of test execution is also very important, but secondary to reliability. Speed is necessary for automated testing in a Continuous Integration environment so developers can gain feedback on their Pull Requests and iterate quickly. A CI system that is slow will cause Pull Requests to pool in a queue, and does not scale as the number of developers increases. The least important criterion is code complexity. This is because developers can accommodate a testing suite that requires more complicated and longer test case, but developers cannot accommodate for the testing suite's speed, reliability, or code coverability. Since this is the most flexible criterion, it is weighted the smallest.

Table 3-1: Evaluation Criteria Weighting

Criterion	**Weighting**
Reliability	40%
Speed	30%
Code Coverage	20%
Code Complexity	10%

3.4 Experimental Procedure

The procedure for obtaining the test results for each quantitative criterion was identical. A set of 10 tests were written in the following categories: database operations, network oper-

ations, configuration changes, mathematics/logic functions, and Android Activity creations. These tests were written in both the Robolectric framework and the Espresso framework. The tests in each category were run 100 times consecutively, and the execution time was tracked, as well as the test success rate. Note that every test case written was verified and in a passing state. Any test failure that occurs during the experiment is due to a random error in the execution of the test, rather than the test case itself.

These test categories were chosen deliberately. Database operations are often hard to test since they require an Android emulator to host the SQLite database. Robolectric overcomes this by storing the database information in virtual memory, and hence, may be a point of inconsistency and warrants experimentation. Network operations in Espresso are handled very differently from Robolectric, and also deserve investigation. A configuration change is when the user rotates their phone from portrait to landscape or vice versa. Configuration changes are also handled very differently between Robolectric and Espresso. Android Activities are individual screens inside an app. The creation of an Activity – or screen – is an expensive operation in terms of time, and thus needs to be tested for performance.

4 Evaluation

The following subsections provide a direct comparison of Robolectric and Espresso against each criterion. The MCDA calculations change the quantitative and quantitative ratings into weighted numerical scores. These scores will be aggregated to determine an overall accepted solution.

4.1 Speed

It is apparent from the experimental data visualized in figure 4-1 that Robolectric executes test cases faster than Espresso. With respect to database operations, network operations, and configuration changes, Robolectric executes tests significantly faster than Espresso. For simpler tasks such as mathematics and logic, as well as Android Activity creations, Robolectric is approximately on par with Espresso. The reason Robolectric is so much faster at the aforementioned operations is because Robolectric does not actually execute those operations. For example, when a test case is saving data to the local Android database, Robolectric will not actually save any data to hard disk. Rather, Robolectric will keep this data in memory, and simulate that this data was saved to disk. With regards to network operations, Espresso will spin up a local web server, and cause network operations to call this server, at which point the local web server will deliver the mocked message. With Robolectric, the network operation does not happen, the data is delivered instantly, and the network operation is simulated. The approach that the Robolectric testing framework has taken of simulating most time-consuming operations has significantly improved the execution speed of automated test cases. In total, it took 95 seconds to run the Robolectric test suite, and 159 seconds to run the Espresso test suite. This will be the parameter used for the MCDA calculation.

4.2 Reliability

It is evident from the experimental data visualized in figure 4-2 that Robolectric executes test cases at a 100% success rate, whereas Espresso executes test cases at a significantly lower success rate. Although math/logic functions did not have a high failure rate in Espresso, every other test category met or exceeded the 5% failure rate threshold specified in the evaluation criteria. Espresso tests intermittently fail because it works by executing lines of test

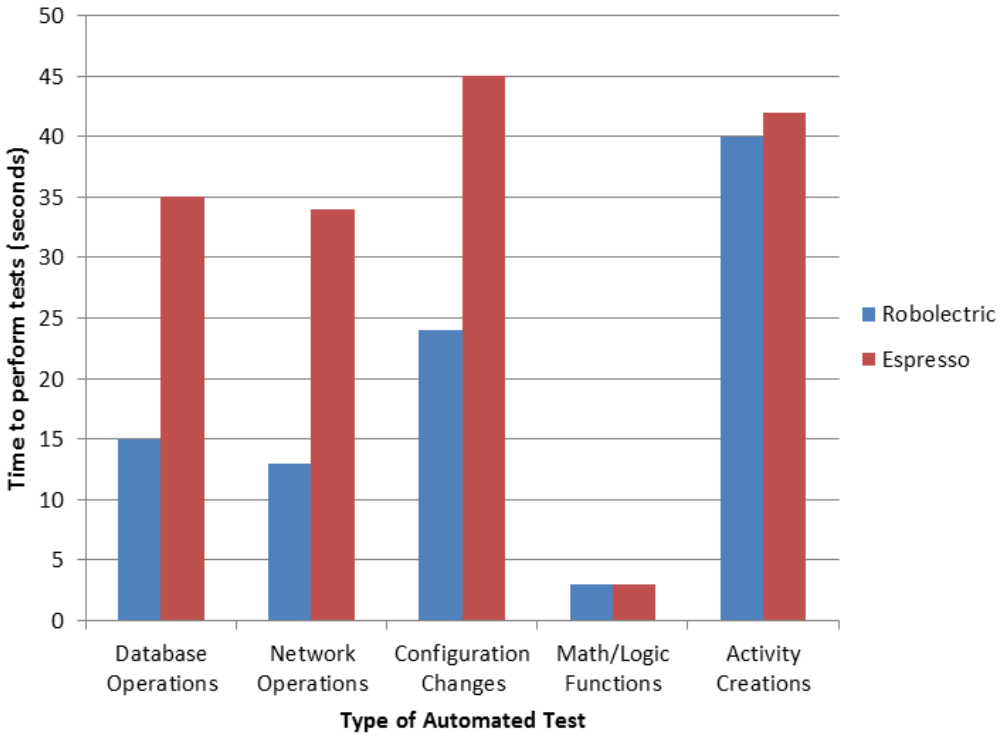

Figure 4-1: Graphical View of Test Execution Speed Analysis

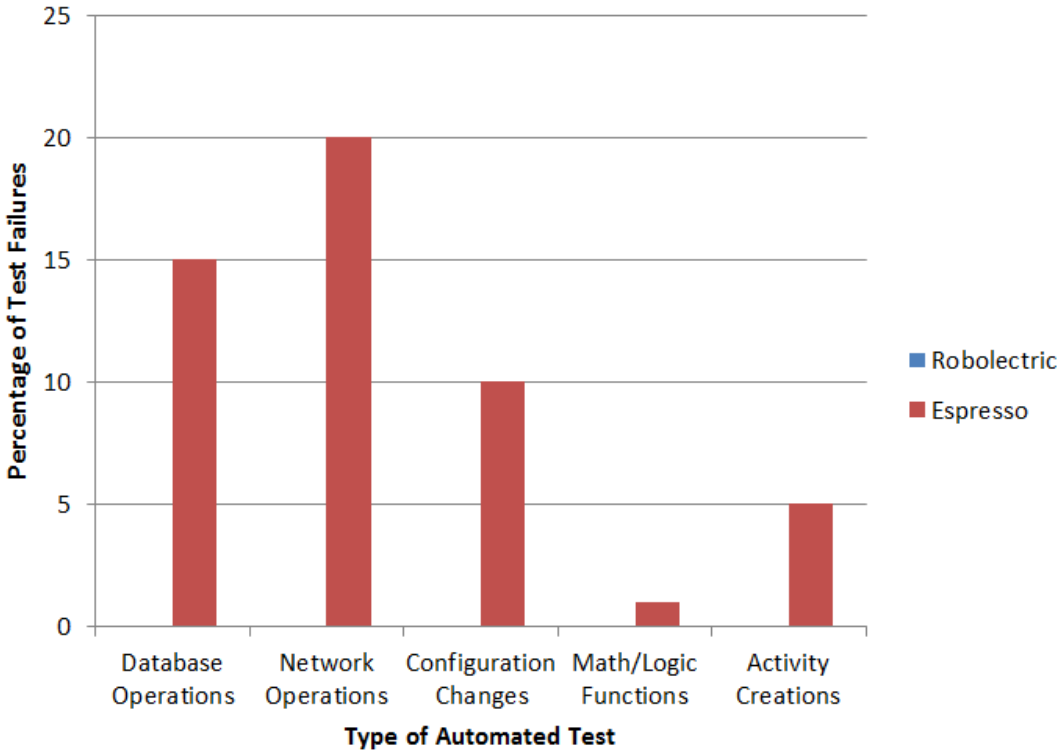

Figure 4-2: Graphical View of Test Reliability Rate

code on a timer. For example, when writing a test that will click on button A, and then click button B, there will be an implicit delay of 100 milliseconds placed between the two button clicks. If the Android OS which is running on the emulator is too slow, button B will not be visible, and the test will fail. Because Espresso works by placing implicit delays, it is known to be unreliable. Unreliability, especially in a Continuous Integration environment, breaks the developers' trust in the automated test suite. A test suite that no one has confidence in is equivalent to having no test coverage at all. Robolectric will receive a perfect score for reliability for the MCDA calculation, and Espresso will be receiving a score of 1/5.

4.3 Code Coverage

Both Espresso and Robolectric provide a high amount of code coverage, however Robolectric has a limitation that Espresso does not. Robolectric performs tests by operating on shadow objects. Shadow objects are Android components which have been stubbed so that every non-essential member function is eliminated. These shadows take a significant amount of time to build, so most developers do not test Android components unless the corresponding shadow object comes prepacked with the version of Robolectric they are using. Google regularly releases new Android components when updating the OS, and whenever Google does this, Pivotal Labs has to keep up by packaging new shadow objects with the next release of Robolectric. This delay causes a perpetual discrepancy between which components are part of the latest Android OS, and which components have been shadowed by the latest version of Robolectric. Espresso has no such limitation, so code coverage on Robolectric isn't as high, relative to Espresso. Espresso will receive favoured scores for this criterion for MCDA.

4.4 Code Complexity

Both Espresso and Robolectric have very similar code complexity regarding number of lines of code, and readability of code. Both testing frameworks use the same set of assertions. That is to say, the syntax used to assert a specific state of the app is exactly the same. Outside of assert statements, both frameworks are founded upon JUnit, and thus both have a `@Before` block and `@After` block for the setup and tear down of the test case. To set up an Espresso test case, you will have to spin up the correct root activity, and then perform clicks on Android components in order to navigate to the correct child activity. To

set up a Robolectric test case, you will have to spin up a default activity, and populate it with a specific Fragment that you want to test. A Fragment is an Android component that corresponds to a single screen inside of an Android Activity. Overall, both frameworks require a different implementation of test cases, but the code length and readability is comparable. They will both be receiving a similar score for the MCDA calculation.

5 Results

The criteria weightings and the subjective/quantitative comparisons serve as the parameters to the Multi-Criteria Decision Analysis calculations. The results of the MCDA yield a set of weighted scores. These scores are summarized in table 5-1.

Table 5-1: Weighted Scores

Criterion	**Robolectric**	**Espresso**
Reliability	0.40	0.08
Speed	0.30	0.18
Code Coverage	0.15	0.20
Code Complexity	0.10	0.10
Total (x 100)	**95**	**56**

Robolectric received an overall score of 95, while Espresso received a score of 56. Robolectric is significantly better than Espresso with regards to reliability and speed. Conversely, Espresso is slightly favourable in terms of code coverage. Overall, the MCDA calculations show that Robolectric is a significantly better Android testing framework for a Continuous Integration environment.

6 Conclusions

The results of the evaluation indicate that Robolectric and Espresso are very different in some aspects and similar in others. The large difference seen in a few of the factors is principally because Robolectric tests run in a Java Virtual Machine, but Espresso tests run in an Android emulator.

In terms of performance, Robolectric wins by a large margin. The test execution speed of tests in Robolectric is significantly higher than its alternative, Espresso. With respect to database operations, network testing, and configuration changes, Robolectric executes tests much faster than Espresso. With respect to mathematics and logic tests, as well as creations of Android Activities, Robolectric and Espresso have approximately the same test execution speed.

Regarding reliability, Robolectric evidently wins against Espresso. Robolectric never provides false test results (it will never report that a test has failed, even though the test is actually in a passing state). However, Espresso reports false test results over 5% of the time, which is enough to say that it is significantly less reliable than Robolectric.

In terms of code coverage of the testing suite, Robolectric will never achieve a similar code coverability to Espresso, simply because of the design of Robolectric. As Google updates Android with more views and components over time, developers of the Robolectric project will have to accommodate these new views by developing shadow objects for these components. Because of this, Robolectric will always lag behind in terms of code coverage.

Regarding code complexity, the difference between Robolectric and Espresso is trivial. Both testing frameworks offer a similar API for writing test cases, so the number of lines of code in each test case, as well as the readability of each test case is virtually the same.

Ultimately, Robolectric's superior speed and reliability for the Pull Request workflow makes it the most appropriate testing framework in a Continuous Integration system. It is relevant to note that even though one cannot achieve full test coverage with Robolectric, it is still the better solution.

7 Recommendations

In light of the results of the analysis, this report recommends that anyone who wants to run automated tests with every Pull Request of an Android app should use the Robolectric testing framework. The increased speed of test execution, as well as the increased test success rate makes it the more appropriate choice for writing automated tests for a Continuous Integration environment. The developers of the Android team should go through whatever training is necessary to get accustomed to the way Robolectric tests are, and should be, written.

To compensate for the fact that Robolectric is primarily a unit testing framework, and not designed to write functional tests out of the box, developers will initially have to spend some time creating a set of abstract tests and helper functions to create a foundation for functional testing. This is not an exceptionally complicated task relative to the production code or automated test suite.

With Robolectric, full code coverage will not be possible to achieve. There will be a discrepancy between the Android views and objects that are available to use in production code, and the Android views and objects that can be testing by Robolectric. This discrepancy can be overcome by developing custom shadow objects, which is a non-trivial solution, and will require a significant amount of engineering time. The technical manager should take these factors into account when estimating timelines and release plans.

References

[1] N. M. Fraser and E. M. Jewkes, *Engineering Economics: Financial Decision Making for Engineers*. Toronto, Canada: Pearson Canada, 2012.

[2] D. T. Milano and P. Blundell, *Learning Android Application Testing*. Birmingham, UK: Packt Publishing, 2015.

[3] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Westford, MA: Prentice Hall, 2008.

Acknowledgements

I would like to thank the Android team at Remind for all the feedback during the completion of this project and guidance on this report.

I used the uw-wkrpt document class written by Simon Law for typesetting.

Glossary

Android Activity A single screen in an Android app that the user can interact with.

Continuous Integration A software development practice where each developer checks in their code to a shared repository several times a day. Before this code can be checked in, the automated tests must be run on the merged version, to ensure code quality and functionality.

Pull Request A proposed set of changes to a software development project's source code. This set of changes will be reviewed by another developer, and then get merged in to the main code base after successful review.