International Journal on New Computer Architectures and Their Applications (IJNCAA) 1(4): 926-941
The Society of Digital Information and Wireless Communications, 2011 (ISSN: 2220-9085)

A Study on the Evaluation of Unit Testing for Android Systems

Ben Sadeh and Sundar Gopalakrishnan
Department of Computer and Information Science,
Norwegian University of Science and Technology,
Trondheim, Norway
{sadeh, sundar}@idi.ntnu.no

ABSTRACT

Unit testing is a method for quickly assessing the building blocks of a program and for obtaining accurate error localizations. However, in order to achieve these qualities, the tests cases need to be isolated, since an external call may imply a connection to a remote database. This requirement also makes unit testing difficult to initiate for classes with outside dependencies, and consequently several approaches have been devised to facilitate unit testing of these methods. This paper focuses on the different ways of unit testing Java methods with external dependencies in an Android application. Additionally, it covers a new category of testing methodology called shadow objects. First, the study examines some of the current methods of testing. Then, it details the different ways a class with an external dependency could be unit tested. Finally, the paper presents a discussion and evaluation of the study.

KEYWORDS

Unit Testing, Integration Testing, Graphical User Interface (GUI), Android Activity, Test-driven Development (TDD), Mock Objects, Robolectric, Shadow Classes

1 INTRODUCTION

The purpose of this study is to promote practices like TDD for the Android platform by examining different ways to unit test methods with external dependencies. While approaches such as mock objects and dependency injections are making it easier to unit test, these methods are often language and system-specific. For this reason, this study is interested in facilitating unit testing for Android. This paper will cover unit testing of methods with external dependencies by specifically looking at GUI methods since they act similarly and for their central role in a mobile application's interaction [1,2].

This paper builds on the authors' previous work, Towards Unit Testing of User Interface Code for Android Mobile Applications [3].

However, unit testing the GUI is difficult [4]. Consequently, several methods have been devised in order to test classes with dependencies in a more practical way [5]. Additionally, as modern GUIs are continuously evolving in complexity, it also becomes harder to establish which parts are relevant to testing [6]. Despite these obstacles, testing the GUI is important for an application's resilience and chance of success and is the basis of this study [7,8].

This paper explores the different methods of unit testing the GUI in an

Android Activity [9]. Section 2 states our motivation and goals for the research and briefly presents some alternate methods for GUI testing an Android activity. Section 3 outlines the steps taken to successfully unit test an Android activity. Then, Section 4 compares the different methods of unit testing to determine which one fits the research goals. Finally, Section 5 concludes the paper with further steps to future research.

2 BACKGROUND AND RELATED WORK

In this research paper, we are interested in unit testing the GUI code of an Android application. Since the testing process is difficult to handle and important for the user experience, this paper has been written with the following research questions in mind:

RQ1. What are the different methods of assessing the GUI code in an Android activity?
RQ2. Is unit testing of the GUI code on the Android platform feasible?
RQ3. If so, is unit testing the GUI code on the Android platform beneficial?

2.1 Testing Concepts

Several testing concepts are relevant to the study and are outlined below.

Android Instrumentation test. Currently, testing the GUI in applications is based on structuring the code in such a way that as much logic as possible is separated from the interface code. Given this distinction, the GUI can be tested using standard instrumentation tests, which are included in the Android SDK.

In Android's own Instrumentation Testing Framework, the framework launches an emulator and runs the application and its test simultaneously, allowing the testing to interact with the whole application. This method can give an accurate depiction of an Android activity's behavior and functionality. However, since this method requires the tests to be run inside an emulator, it performs slow and is difficult to isolate.

Dalvik Virtual Machine (VM). The Dalvik VM is a java bit code interpreter used by Android mobile devices. It is optimized in terms of battery life and watt efficiency [10,11]. During Android Instrumentation tests, an emulator simulates a Dalvik VM environment to achieve a behavior very close to a real Android device [12]. However, since the program code needs to pass through an interpreter that resides inside an emulator, testing realism comes at the cost of testing speed. Because speed is of importance to the study, this paper may favor methods of testing that circumvent the Dalvik VM and use the standard Java VM instead.

Mock Objects. When working with automated software testing, unit testing allows developers to quickly assess critical boundaries of their applications such as upper and lower limits and corner-cases. However, in order to maintain high testing speeds, unit tests need to adhere to several conditions including being isolated from potentially expensive external calls. To facilitate in testing of modules that do have external dependencies, programmers may

Figure 1: The calculator application with the add and subtract function

substitute external calls of the code with mock objects to make the assessments non-deterministic. For example, a method that interacts with a database may instead call a static database that always returns the same answers. Depending on the functionality in question, the method can now be properly unit tested because the external dependency has been isolated.

On the other hand, using too many mock objects can lead to more lines of code that needs to be maintained in order to facilitate testing. While mock objects may solve the issue of dependency isolation, they can introduce other inconveniences such as test methods maintenance.

3 SUGGESTED TESTING APPROACH

An essential part of GUI code is to interact with the graphical components on the screen, such as buttons and text fields. A well-designed application separates the GUI code from the business logic. For example, a controller's job is to receive interactions from the user, such as a button click, and react to the interaction, perhaps involving requests to the business logic.

Unit testing a controller in such an application is challenging, but possible with commonly used techniques for unit testing business logic [13]. This section will take advantage of certain programming techniques using a simple example application containing a method in the controller class to be tested. The approach involves breaking the dependencies to the user interface framework, and optionally to the business logic. In Subsection 3.1 the example application and the method to be tested are described. Then, Subsection 3.2 covers the steps taken to unit test the method using the standard Eclipse environment. Finally, Subsection 3.3 outlines the convenience of unit testing using an assisting framework.

3.1 Example Application

Listing 1: Original `onClick()` implementation

```
public void onClick(View view) {
  // Get the token that 'view' maps to
  CalculatorButton button = CalculatorButton.findById(view.getId());
  calculatorState.pushToken(button);
  updateDisplay();
}
```

The example program will be a custom made calculator. It supports addition and subtraction of numbers, and has a user interface similar to traditional pocket calculators, as illustrated in **Figure 1**.

The calculator contains three main classes that are illustrated in **Figure 2**.

CalculatorButton. This is an enumerator with one value for each button of the calculator. It maps an Android component ID to the CalculatorButton value that is used to represent said button. The '+' button in the user interface maps, for example, to `CalculatorButton.B\_add`.

CalculatorState. This class handles the business logic of the calculator. It accepts a `CalculatorButton` as its input and handles the state of the calculator when numbers are received.

Calculator. This class is the Android Activity of the application. It handles the user interaction by listening to click events in the user interface, accomplished by the `onClick()` method as shown in **Listing 1**.

`OnClick()`. This method performs a two-way communication with the user interface: It retrieves the button clicked and updates the display, and to do this correctly, it needs to interact with the business logic.

`UpdateDisplay()`. This simple

Listing 2: CalculatorClickListener

```
 class RealCalculatorClickListener {
  public void onClick(View view) {
    // Definition omitted
  }
}

class CalculatorClickListener extends RealCalculatorClickListener
    implements OnClickListener {
  // Empty class
}
```

Listing 2: ViewIdGetter

```
class ViewIdGetter {
  int getId(View view) { return view.getId(); }
}

class RealCalculatorClickListener {

  private ViewIdGetter viewIdGetter;
  RealCalculatorClickListener(ViewIdGetter viewIdGetter) {
    this.viewIdGetter = viewIdGetter;
  }

  public void onClick(View view) {
    int viewId = viewIdGetter.getId(view);
    // Remainder of definition omitted
  }
}
```

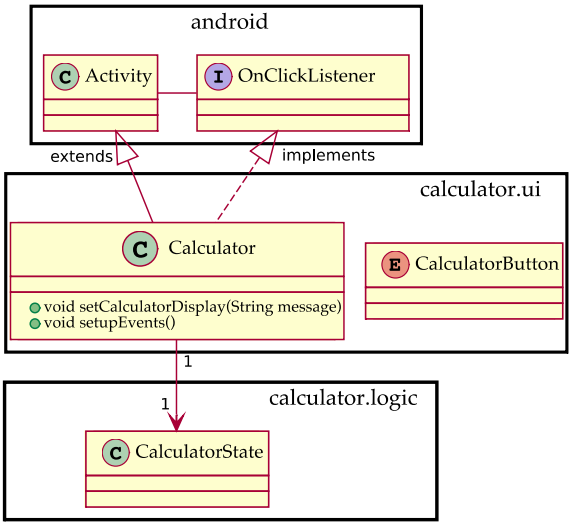

Figure 2: The main classes in the calculator application before testing

method will be tested using the same techniques as `onClick()`.

3.2 Standard Environment Approach

In this approach, the default Eclipse environment is considered for the Android development [14]. However, out of the box, it doesn't grant access to the Android classes, and so it is not possible to initialize the GUI classes such as the Calculator class.

3.2.1 Avoiding Initializing Classes

By extracting the `onClick()` method into a different class, say `CalculatorClickListener`, the code can be tested without initializing `Calculator`. If `CalculatorClickListener` implements the `OnClickListener` interface, it can act as the click listener for `Calculator`, but this prevents `CalculatorClickListener` from being instantiated. Therefore, the proposed approach works around the issue by creating a class that inherits from the class that implements `onClick()`, as shown in Listing 2.

The proposed approach instantiates `RealCalculatorClickListener` in the unit test. `CalculatorClickListener` is not supposed to contain any code, and therefore it should not require testing. However, in this implementation, `RealCalculatorClickListener` takes arguments in its constructor, meaning that `CalculatorClickListener` must have a constructor as well.

Since Android classes cannot be instantiated in this environment, any classes extending or implementing them cannot be tested. Therefore, the constructor of `CalculatorClickListener` remains untested.

3.2.2 Interacting with Android Components

Code that interacts directly with Android classes, such as `onClick()`, cannot run in a unit test because they cannot be instantiated. The solution in the standard environment is to extract the code that performs the interaction into a separate class, which then can be faked in the unit test, as illustrated in **Listing 2**.

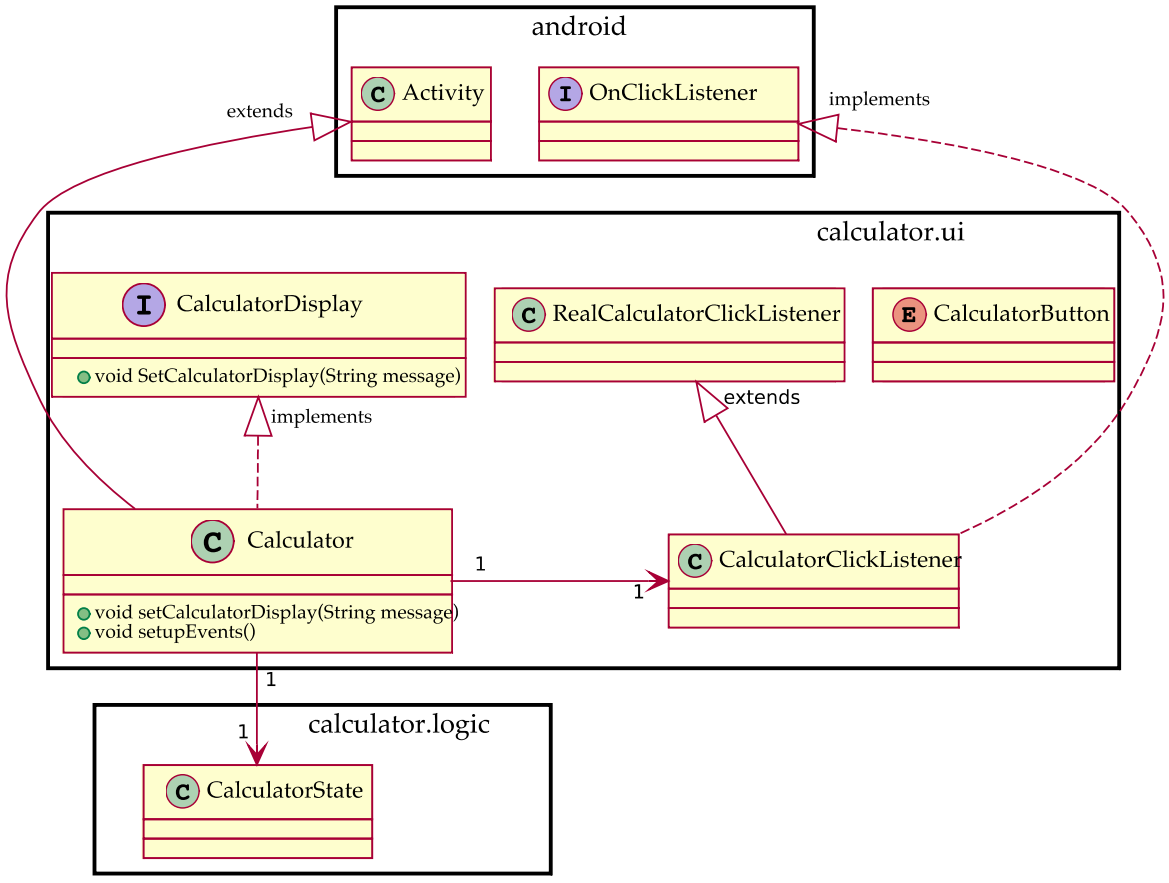

Figure 3: The main classes in the calculator application before testing

This leaves `ViewIdGetter.getId()` untested because it requires a View instance, and by extracting similar statements, one is able to minimize and isolate the untested code. **Figure 3** provides an overview of the calculator classes after the refactoring. `onClick()` can now be unit tested using fake objects, as shown in **Listing 3**.

3.3 Robolectric Approach

Robolectric [15] is an open-source framework built to assists in unit testing and is released under the open-source MIT license. The framework is comprised of a series of mock objects that mimic many of the actual Android classes, several of which are unable to be initialized conventionally because of dependencies to the Dalvik VM. However, Robolectric proposes a new way to relate to mock objects through something they dub 'shadow classes.' Instead of re-writing the program code to interact with their mock objects, Robolectric intercepts the code during test-time and refers the appropriate methods to their shadow equivalent. In this case, the shadow object 'shadows' a real object, so that the code will call different classes during run-time and test-time, circumventing the need to change the code for testing purposes.

Listing 3: Testing the CalculatorClickListener

```
public class CalculatorClickListenerTest {

  static class FakeCalculatorDisplay implements CalculatorDisplay {
    public String display;
    public void setCalculatorDisplay(String message) {
      display = message;
    }
  }

  static class FakeViewIdGetter extends ViewIdGetter {
    public static final CalculatorButton CLICKED_BUTTON =
        CalculatorButton.B_05;
    int getId(View unused) { return CLICKED_BUTTON.getId(); }
  }

  static class FakeCalculatorState extends CalculatorState {
    public CalculatorButton receivedToken;
    public static final String DISPLAY =
"Display:FakeCalculatorState";

    public void pushToken(CalculatorButton button) {
      assertEquals(null, receivedToken);
      receivedToken = button;
    }

    public String getDisplay() { return DISPLAY; }
  }
```

Listing 4: Continued

```
  private RealCalculatorClickListener calculatorClickListener;
  private FakeCalculatorState calculatorState;
  private FakeCalculatorDisplay calculatorDisplay;
  private FakeViewIdGetter viewIdGetter;

  @Before
  public void setUp() {
    calculatorState = new FakeCalculatorState();
    calculatorDisplay = new FakeCalculatorDisplay();
    viewIdGetter = new FakeViewIdGetter();
    calculatorClickListener = new RealCalculatorClickListener(
        calculatorState, calculatorDisplay, viewIdGetter);
  }

  @Test
  public void testOnClick() {
    calculatorClickListener.onClick(null);
    assertEquals(FakeViewIdGetter.CLICKED_BUTTON,
        calculatorState.receivedToken);
    assertEquals(FakeCalculatorState.DISPLAY,
        calculatorDisplay.display);
  }
}
```

As was explained in subsection 3.2, in order to unit test `Calculator` in the standard environment, the code had to be refactored to avoid initializing the Android framework classes. Conversely, by using the Robolectric framework, the `Calculator` class can be tested with no refactoring, as illustrated in **Listing 5**.

4 EVALUATION AND DISCUSSION

The Calculator application was successfully unit tested in the standard environment, but only after a significant amount of refactoring and boilerplate code. This approach may become unmanageable for larger applications as refactoring of the methods may grow to be complicated to maintain and difficult to debug.

However, Robolectric's Shadow Classes made it easy to write unit tests by shadowing the real classes and bypassing the need for extra steps and abstractions.

This study aims for efficiency in unit testing the GUI code in an Android mobile application. By making use of the Robolectric framework, certain qualities that are important to this research can be achieved. This paper both aspires for and achieves:

- tests that run fast
- tests that are relevant
- code that is easy to maintain

Based on the initial research questions and the qualities listed above, there are several categories of software tests that are of interest.

4.1 Automated Software Testing Categories

Listing 5: Testing Calculator using the Robolectric framework

```
public class CalculatorTest {

    @Test public void testOnClick() {
        Calculator calculator = new Calculator();
        calculator.onCreate(null);

        View fakeView = new View(null) {
            @Override public int getId() {
                return CalculatorButton.B_04.getId();
            }
        }


        calculator.onClick(fakeView);
        TextView display = (TextView)calculator.findViewById(
                R.id.CalculatorDisplay);
        assertEquals("4.0 ", display.getText());
    }
}
```

The following section will differentiate between the different testing methods and explain the author's reason for choosing one over the other.

a) *Unit Testing*. To ensure that the individual components in a program are working, one needs to assess that the smallest building blocks are built correctly. As a result, Unit tests [16,17] are run on individual functions and in some cases even whole classes in isolation from the rest of the application. Thus, design guidelines and techniques for breaking dependencies have been developed. For example, combination of the Dependency Injection design pattern [18] and Mock Objects can be used to allow unit testing of a class with dependencies that would otherwise make it hard to test.

Similarly, unit testing a GUI is similar to testing a class with several external dependencies, because the interaction with a GUI framework represents a black-box to the unit test.

Because unit tests cover specific parts of the program, they offer the advantage of running quickly and independent of the rest of the application.

b) *Integration Testing*, *Limitations*. After the different components have been tested, they can be put together to see whether they perform as expected.

Integration testing [19] is performed by combining multiple parts of the application and is useful for checking that the different parts of the program are working together.

Integration testing is relevant for quality assurance since it covers larger parts of the program. However, these tests may run slower due to some times unforeseen dependencies and so they do not meet the conditions set forth by this paper.

Figure 4 and Figure 5 illustrates the difference between a unit test and an integration test.

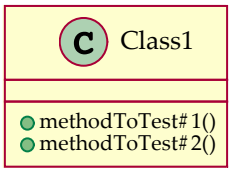

Figure 4: Unit testing: a single isolated component is tested

4.2 Test Results

The `onClick()` method was tested[1] using three different methods, summarized in **Table 1**. Furthermore, comparison of the methods in relation to the research goals is illustrated in Table 2.

Table 1. Summarization of test approaches for the Calculator application

Method	Type of test	Test runtime
Android Instrumentation	Integration test	5.629 sec
Standard environment	Unit test	0.69 sec
Robolectric	Unit test	1.16

Table 2. Comparison between the selected methods

Factors	Android Instrumentation (Integration test)	Standard environment (Unit test)	Robolectric (Unit test)
Ease of writing tests	++	-	+
Ease of maintenance	+	--	+
Error localization	--	-	++
Relevance	+	+	+
Speed	--	++	+

[1] Computer specifications:
Intel Core 2 Duo E7500, 4 GB RAM, Debian GNU/Linux, Eclipse Helios

The notation is explained in the following table:

++	stands for	very good
+	"	good
-	"	unsatisfactory
--	"	very unsatisfactory

For applications that are more complex, using the Robolectric framework is likely to be more practical, because it scales by allowing developers to avoid having to maintain a collection of fake objects and interfaces.

Because of its nature, standard Unit testing will remain as the quickest testing method. However, classes that are refactored to allow for unit testing make them more difficult to maintain correctly. By using the Robolectric framework, one can achieve close to the speed of unit tests together with the ease of writing found in the integration tests.

From Table 1 and Table 2, Robolectric and Android Instrumentation can be used for testing user interface code for the number of qualities listed in it. Again, we conducted a controlled experiment with the two above alternatives for the same simple calculator application. Four people participated for this experiment, two from an academic background and two from an industry background. This experiment is conducted to find the preferable method in terms of TAM [20] model, as illustrated in Figure 6.

The questionnaire experiment was carefully prepared to reflect three factors for the Technology Acceptance Model (TAM) viz., Perceived Ease Of Use (PEOU), Intention to Use (IU) and

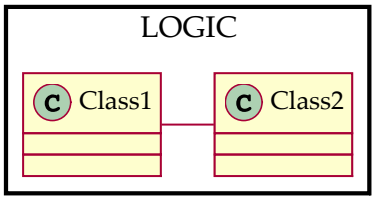

Figure 5: Integration testing: Interaction between two or more components is tested

perceived usefulness. The participants were asked to use these two alternative testing methods for the calculator application and then to fill out a questionnaire on their experience with this experiment. We assign value 4 to completely agree and 0 to completely disagree with the statement in the questionnaire. The questionnaire used for this experiment is provided as an appendix for reference.

Based on the experiments results, the evaluation results are plotted in Figure 7. As per the participants view, Robolectric considerably outperforms Android Instrumentation testing alternative in all the three factors, especially for the alternative 'Intention to use.'

However, the Robolectric approach is not a complete replacement for instrumentation tests, as it does not test the actual graphical components.

Moreover, responsibility for the correct assessments is given to Robolectric, and the developer needs to be mindful over the fact that the Android classes are untouched and that the shadow classes do the actual work.

Lastly, shadow classes are written independently and without automation, so that when Android framework is updated with new classes, there is no automated process of including the new additions. Therefore, Robolectric currently plays a cat and mouse game by supporting the latest Android framework ad hoc.

4.3 Threats to Validity

Wohlin et al. suggests four categories for threats to validity in experiments: conclusion validity, construct validity, internal validity and external validity. To identify the best alternative testing methodology, we conducted the controlled experiment through practitioners from academic and industry as explained in previous section.

Conclusion validity concerns the relationship between the treatment given and the outcome in measured variables. One important question is whether the sample size is big enough to justify the conclusions; here we have taken only four participants, two academics and two from industry with solid technical background. So the effect low sample size in terms of conclusion validity is minimized.

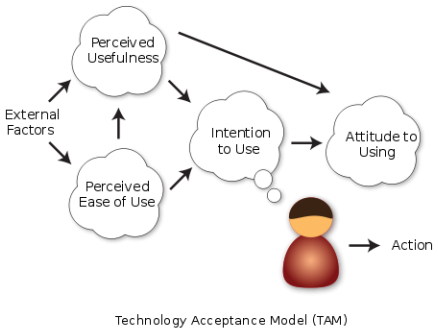

Figure 6: Illustrations of TAM model [20]

Construct validity is concerned with the inference from the measures made in the experiment to the theoretical constructs to be observed. This controlled experiment is conducted for simple applications. Naturally, more experiments with a wider range of experimental tasks would be necessary to draw more certain conclusions for practical usage.

Internal validity means that the observed outcomes were due to the treatment, not to other factors i.e., only because of the two testing methodologies. Participants were new to both testing methodologies in the experiment and hence no bias effect towards any approach.

External validity questions whether it is possible to generalize from the experimental setting to other situations. This is impossible to answer from the experimental data, but intuitively there is no particular reason why the situation should be different for other application since we constrained ourselves only to Android development.

5 CONCLUSION AND FUTURE WORK

This paper explores the different options developers have for assessing the correctness of their Android mobile application.

A GUI component was successfully unit tested by adding extra code and abstractions.

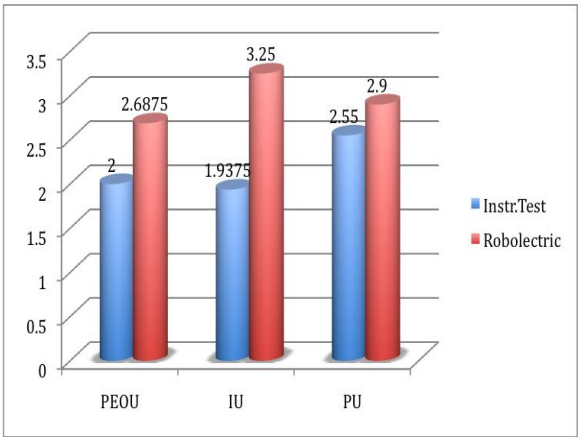

Figure 7: Evaluation by TAM factors

Robolectric allowed tests to be written to said component with less refactoring of the original source code, and the resulting tests were fast and provided relevant test coverage of the GUI code. For this reason, unit testing GUI code is likely to benefit Android developers.

A controlled experiment with a simple application was conducted in order to establish the best testing approach between Android Instrumentation testing and Robolectric. The study showed that the participants preferred Robolectric to conventional instrumentation testing.

Our research conclusion currently only applies to our example application, and in future studies, we wish to expand test coverage to larger programs to obtain additional confidence in recommending unit testing with Robolectric for more complex applications and systems.

6 REFERENCES

1. Liu Zhifang, Liu Bin, and Gao Xiaopeng. Test automation on mobile device. In Proceedings of the 5th Workshop on Automation of Software Test, AST '10, pages 1–7, New York, NY, USA, 2010. ACM.
2. Sun-Myung Hwang and Hyeon-Cheol Chae. Design & implementation of mobile GUI testing tool. In Proceedings of the 2008 International Conference on Convergence and Hybrid Information Technology. IEEE Computer Society, 2008.
3. Ben Sadeh, Kjetil Ørbekk, Magnus Eide, Njaal Gjerde, Trygve Tønnesland and Sundar Gopalakrishnan. Towards Unit Testing of User Interface Code for Android Mobile Applications In Proceedings of the 2011 Communications in Computer and Information Science, Software Engineering and Computer Systems.
4. P. Hamill. Unit Tests Framework. O'Reilly, 2004.
5. Penelope Brooks, Brian Robinson, and Atif M. Memon. An initial characterization of industrial graphical user interface systems. In ICST 2009: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation.
6. Kai-Yuan Cai, Lei Zhao, Hai Hu, and Chang-Hai Jiang. On the test case definition for GUI testing. In Quality Software, 2005. (QSIC 2005). Fifth International Conference on, sept 2005.
7. Atif M. Memon. A comprehensive framework for testing graphical user interfaces. Ph.D., 2001.
8. Alex Ruiz and Yvonne Wang Price. Test-driven GUI development with testng and abbot. Software, IEEE, 24(3):51 –57, may-june 2007.
9. Google inc. Android activity, 2011. Available from: http://developer.android.com/reference/android/app/Activity.html [cited 2011-03-09].
10. David Ehringer, The Dalvik Virtual Machine Architecture, March 2010. Available from: http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf [cited 2011-07-04].
11. Google inc, Android Architecture, 2011. Available from: http://developer.android.com/guide/basics/what-is-android.html [cited 2011-07-04].
12. Google inc. Testing fundamentals, 2011. Available from: http://developer.android.com/guide/topics/testing/testing_android.html [cited 2011-03-09].
13. Michael Feathers. Working Effectively with Legacy Code. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

14. Google inc. Android developing introduction, 2011. Available from: http: //developer.android.com/guide/developing/index.html [cited 2011-03-09].
15. Pivotal Labs. Robolectric, 2011. Available from: http://pivotal.github.com/robolectric/ [cited 2011-03-09].
16. IEEE 1008 - IEEE standard for software unit testing, 1987.
17. R.S. Freedman. Testability of software components. IEEE Transactions on Software Engineering, 17:553–564, 1991.
18. Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA, 1999.
19. Ursula Linnenkugel and Monika Müllerburg. Test data selection criteria for (software) integration testing. In Proceedings of the first international conference on systems integration on Systems integration '90, 1990.
20. Davis, F. D. (1989), "Perceived usefulness, perceived ease of use, and user acceptance of information technology", *MIS Quarterly* 13(3): 319–340

Appendix

Below is the questionnaire used for our experiment to identify the best practice testing methodology. The experiment was conducted with four practitioners, two from an academic background and two from an industry background. They are provided with a simple calculator application to be used with both Android Instrumentation testing and the Roboelectric framework. After trying both alternatives, the below questionnaire was filled out.

Sl No	Questionnaire	Testing	Comp. Agree	Partly Agree	Neither/Nor Agree/disag.	Partly Disagr.	Comp. Disagr.
1	Method gave me a better understanding of the unit testing. (PEOU)	Android Instr. test					
		Roboelectric					
2	I found this method is very easy to master. (PEOU)	Android Instr. test					
		Roboelectric					
3	I found very easy to use and recognize this testing. (PEOU)	Android Instr. test					
		Roboelectric					
4	I was not often confused about how to apply this testing to android mobile UI application. (IU)	Android Instr. test					
		Roboelectric					
5	If I need to test UI in android mobile application in a future project, I would use this testing.(IU)	Android Instr. test					
		Roboelectric					
6	I will try this method if I been assigned in my future work involving mobile application. (IU)	Android Instr. test					
		Roboelectric					
7	If I am working as freelance consultant for a customer who needs help testing applications (mobile UI)that are performed in android system, I would use this notation in discussions with that customer. (IU)	Android Instr. test					
		Roboelectric					
8	If I am employed in a company which discusses what method to test android mobile UI and someone suggest this method, I would support that.(IU)	Android Instr. test					
		Roboelectric					
9	The method made Unit testing User Ineterface more systematic in android.(PU)	Android Instr. test					
		Roboelectric					
10	It is very easy to get used this method in a project. (PU)	Android Instr. test					
		Roboelectric					

11	I can read and understand this method quickly. (PU)	Android Instr. test					
		Roboelectric					
12	This method is easy to remember. (PU)	Android Instr. test					
		Roboelectric					
13	This method made me more productive in testing where the UI applications developed with android.(PU)	Android Instr. test					
		Roboelectric					