# Towards Unit Testing of User Interface Code for Android Mobile Applications

Ben Sadeh, Kjetil Ørbekk, Magnus M. Eide, Njaal C.A. Gjerde, Trygve A. Tønnesland, and Sundar Gopalakrishnan

Department of Computer and Information Science,
Norwegian University of Science and Technology,
Trondheim, Norway
{sadeh,orbekk,magnuei,njaalchr,
tonnesla}@stud.ntnu.no,
sundar@idi.ntnu.no

**Abstract.** As the availability and popularity of mobile applications grows, there is also an increased interest for them to be solid and well tested. Consequently, there is also an interest in assessing the correctness of their system rapidly, since smart phone applications usually develop quickly and have a lower lifecycle as compared to desktop applications. We are specifically interested in an efficient way of testing the Graphical User Interface (GUI), as it is both central to the user experience and harder to evaluate than standard business logic. This research paper is a study on the different ways to assess the validity of the GUI code for an Android mobile application with special focus on unit testing. It describes the available testing techniques and details the difficulty in writing unit tests for GUI code. Finally, the study gives a recommendation based on the different testing approaches available, followed by a discussion of both the implications and limitations of the findings.

**Keywords:** Unit testing, Integration testing, GUI, Android application, Robolectric, Model-View-Controller (MVC), Model-View-ViewModel (MVVM), Test-driven development (TDD).

## 1 Introduction

The smart phone market is seeing a stable growth, with more and more people depending upon their mobile devices to manage their email, entertainment, shopping and scheduling. However, quick development processes of mobile applications may come at the cost of quality assurance [1].

This study proposes a testing approach for Android mobile applications that recognizes and fits the smartphone's fast-paced development cycle by focusing on the GUI and unit testing [2].

J.M. Zain et al. (Eds.): ICSECS 2011, Part III, CCIS 181, pp. 163–175, 2011.

However, unit testing the GUI is difficult [3]. Consequently, several methods have been devised in order to test classes with dependencies in a more practical way [4]. Additionally, as modern GUIs are continuously evolving in complexity, it becomes harder to establish which parts are relevant to testing [5]. Nevertheless, testing the GUI is important for an application's resilience and chance of success [6,7].

This paper explores the different methods of assessing the GUI in an Android Activity [8] in special relation to unit testing. 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 possible future research.

## 2 Background and Related Work

In this research paper we are interested in unit testing the GUI code of an Android mobile application. Since the testing prosess 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, or does instrumentation testing suffice?

### 2.1 Testing Methodologies

**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 Software development kit (SDK).

In Android's own Instrumentation Testing Framework [9], the framework launches an emulator and runs the application and its test simultaneously, allowing the testing to interact with the whole application. Consequently, these instrumentation tests can be classified as integration tests.

Since this method requires the tests to be ran inside an emulator, it performs slower while being more difficult to isolate.

**Model-View-ViewModel.** The MVVM pattern [10,11] uses data bindings to separate the GUI code from the GUI components. This software architecture fits our research goals for unit testing GUI code, but is currently not supported on the Android platform.

## 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 MVC application separates the GUI code from the business logic. In this case, the 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 [12]. This section will take advantage of mentioned techniques using a simple example program 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

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.


**Fig. 1.** The calculator application with the add and substract functions

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. For example, the '+' button in the user interface maps to CalculatorButton.B_add.

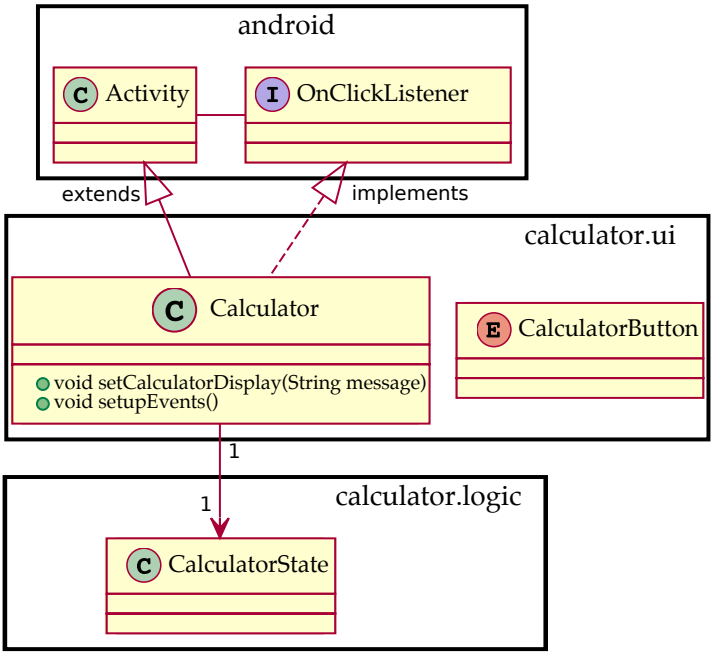



**Fig. 2.** The main classes in the calculator application before testing

**Listing 1.1.** 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();
}
```

**CalculatorState.** The business logic is handled by this class. 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, done by the onClick() method as shown in Listing 1.1.

OnClick() 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() is a simple method that was 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 [13]. However, out of the box it doesn't permit access to any of the Android classes, and so it is not possible to initialize the GUI classes such as the Calculator class.

**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. Consequently, the proposed approach works around the issue by creating a class that inherits from the class that implements onClick(), as shown in Listing 1.2.

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.

**Interacting with Android Components.** Code that interacts directly with Android classes, such as onClick(), cannot run in a unit test because they cannot

**Listing 1.2.** CalculatorClickListener

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

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

**Listing 1.3.** 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
  }
}
```

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 1.3.

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 1.4.

### 3.3 Robolectric Approach

In order to unit test Calculator in the standard environment, we had to refactor the code and avoid initializing the Android framework classes. Alternatively, we could provide our own replacement definitions for the Android classes that would be possible to initialize. Fortunately, this effort has already been made by Pivotal Labs in an open framework called Robolectric [14].

Robolectric provides lightweight mock versions of the Android framework classes, called shadow objects. Moreover, they can be initialized and used in conjunction with unit tests. For example, setting the contents of a Robolectric TextView allows for retrieving the same contents during a test.

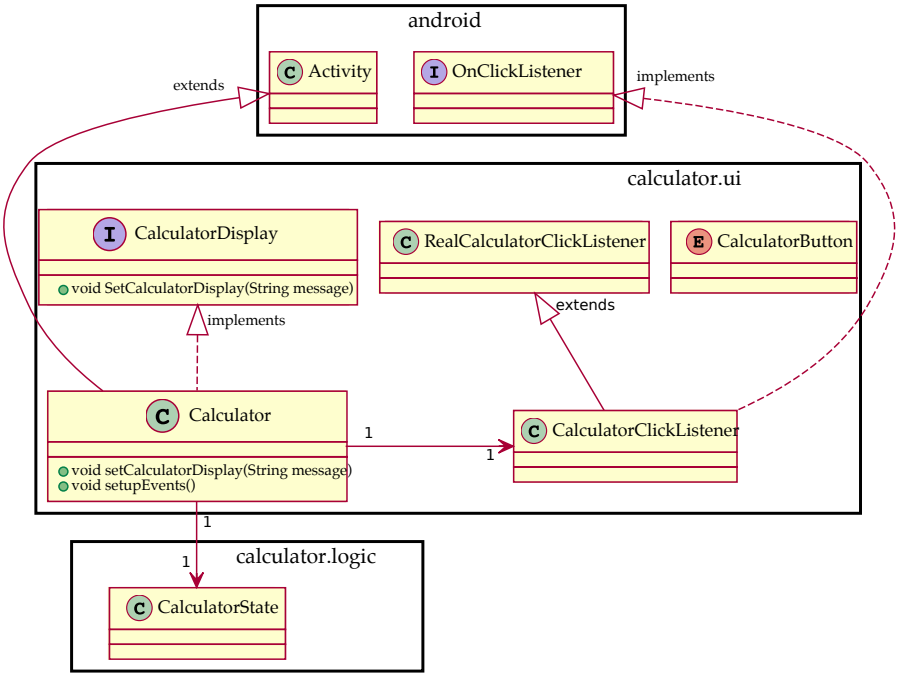



**Fig. 3.** The main classes in the calculator application after testing

**Listing 1.4.** 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; }
  }


  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);
  }
}
```

**Listing 1.5.** 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());
    }
}
```

Consequently, by using the Robolectric framework, the Calculator class can be tested with no refactoring, as illustrated in Listing 1.5.

## 4 Results and Discussion

The Calculator application was successfully unit tested in the standard environment, but only after a significant amount of refactoring and boilerplate code. Therefore, this approach may become unmanageable for larger applications.

However, the Robolectric framework makes it easy to write unit tests by requiring fewer extra steps and abstractions.

This study is 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 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 list above, there are several categories of software tests that are of interest.

### 4.1 Automated Software Testing Categories

*a) Unit Testing*

To ensure that the individual components in a program are working, we need to assess that the smallest building blocks are built correctly. As a result, Unit tests [15,16] are run on individual functions and some times 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 [17] and fake 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.

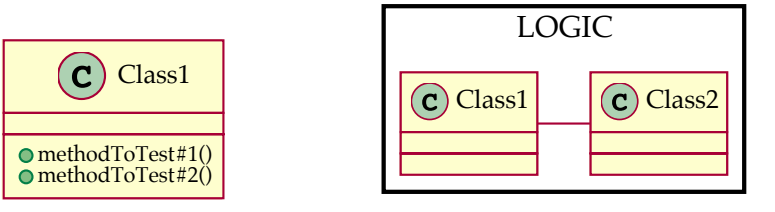
*b) Integration Testing, Limitations*

After the different components have been tested, they can be put together to see whether they perform as expected.

Integration testing [18] 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 often relevant since it covers larger parts of the program. However, these tests run slower due to the added dependencies, especially when ran inside an emulator.

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




(a) Unit Testing: One isolated component is tested

(b) Integration Testing: Interaction between two or more components is tested

**Fig. 4.** Illustration of Unit and Integration testing

.

### 4.2 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.

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

**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 sec |

**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 | − − | ++ | + |

The notation is explained in the following table:

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

For more complex applications, using the Robolectric framework is likely to be more practical, because it allows developers to unit test their classes in isolation without having to maintain a collection of fake objects and interfaces.

Because of it's nature, standard Unit testing will remain as the quickest testing method. However, classes that are refactored to allow for unit testing makes them difficult to maintain correctly. On the other hand, Integration tests are well supported and simple to run, but lack the speed and error localization that unit tests have. By using the Robolectric framework, one can achieve the speed of unit tests together with the ease of writing found in the integration tests.

However, the Robolectric approach is not a complete replacement for instrumentation tests as it does not test the actual graphical components. Moreover, this recommendation depends on the Robolectric framework to be written correctly, as it assumes responsibility for returning the accurate assessment.

## 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 sucessfully unit tested by adding extra code and abstractions.

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.

Our research 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.

## References

1. Zhifang, L., Bin, L., Xiaopeng, G.: Test automation on mobile device. In: Proceedings of the 5th Workshop on Automation of Software Test, AST 2010, pp. 1–7. ACM, New York (2010)
2. Hwang, S.M., Chae, H.C.: Design & implementation of mobile GUI testing tool. In: Proceedings of the 2008 International Conference on Convergence and Hybrid Information Technology. IEEE Computer Society Press, Los Alamitos (2008)
3. Hamill, P.: Unit Tests Framework. O'Reilly, Sebastopol (2004)
4. Brooks, P., Robinson, B., Memon, A.M.: 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 (2009)
5. Cai, K.Y., Zhao, L., Hu, H., Jiang, C.H.: On the test case definition for GUI testing. In: Fifth International Conference on Quality Software, QSIC 2005 (September 2005)
6. Memon, A.M.: A comprehensive framework for testing graphical user interfaces. Ph.D (2001)
7. Ruiz, A., Price, Y.W.: Test-driven GUI development with testng and abbot. IEEE Software 24(3), 51–57 (2007)
8. Google Inc. Android activity, (2011), `http://developer.android.com/reference/android/app/activity.html` (cited 2011-03-09)
9. Google Inc. Testing fundamentals, (2011), `http://developer.android.com/guide/topics/testing/testing_android.html` (cited2011-03-09)
10. Reenskaug, T.M.H.: Models - views - controllers (1979), `http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf` (cited 2011-03-09)
11. Feldman, A., Daymon, M.: WPF in Action with Visual Studio 2008. Manning Publications Co., Greenwich (2008)
12. Feathers, M.: Working Effectively with Legacy Code. Prentice Hall PTR, Upper Saddle River (2004)
13. Google Inc. Android developing introduction (2011), `http://developer.android.com/guide/developing/index.html` (cited 2011-03—09)
14. Pivotal Labs. Robolectric (2011), `http://pivotal.gitthub.com/robolectric/` (cited 2011-03-09)

15. IEEE 1008 - IEEE standard for software unit testing (1987)
16. Freedman, R.S.: Testability of software components. IEEE Transactions on Software Engineering 17, 553–564 (1991)
17. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
18. Linnenkugel, U., Müllerburg, M.: Test data selection criteria for (software) integration testing. In: Proceedings of the first international conference on systems integration on Systems integration 1990, pp. 709–717 (1990), http://portal.acm.org/citation.cfm?id=93024.93262