Automated GUI Testing on the Android Platform[1]

With increasing graphical capabilities of today's mobile phones, an easy to use GUI has become a critical success factor for mobile software applications. As a consequence of the more advanced GUI capabilities, manual GUI testing has become more complex and error prone. Thus it is absolutely critical for efficient mobile application development to establish automated GUI testing. For Google's open source platform Android there exist various GUI testing frameworks for mobile applications. In this paper we describe and analyze two approaches for testing mobile GUI applications: the Android Instrumentation Framework, and the Positron Framework. We will also show the strength and weaknesses of both approaches.

Martin Kropp, Pamela Morales | **martin.kropp@fhnw.ch**

Introduction

In software development, the correct behavior and operation of the graphical user interface has become a critical success factor for the acceptance of GUI-centric applications. This is even more important for applications on mobile devices with their limited display size and their special input devices, like fingers, sticks, or small keyboards. Thus testing the correct functionality of the GUI in an application is very critical. Because of the inherent limitations of manual GUI testing there is a strong need for automated testing concepts for mobile GUI applications. However, automated GUI testing for mobile apps is still at its beginning.

Android, Google's software stack for mobile applications, provides two approaches for automated GUI testing: The *Android Instrumentation Framework* and the *Positron Framework*. The goal of this paper is (1) to show how to write GUI tests with both approaches by example, (2) and to analyze the strength and weaknesses of the two approaches.

After a short introduction to automated GUI testing in general, we present the Android Instrumentation Framework and the Positron Framework. We continue with a comparison and an analysis of both approaches and conclude with an outlook on further work.

Automated GUI Testing

GUI testing is the process of testing an application with a graphical user interface to ensure correct behavior and state of the GUI. This includes verification of data handling, control flows, states, display of windows and dialogs, for example. It verifies the correct interaction of GUI components with the user [Ger97].

Testing GUI applications in general rises special challenges. The event-driven nature of GUIs presents a serious difficulty; the user can click anywhere on the screen. For mobile GUI apps, another important issue is that the target platform is different from the development platform. These circumstances make mobile GUI more difficult compared to pure functional testing [Mar98]. Manual GUI testing is very error prone and hardly reproducible, causing extremely high effort. Providing automated testing for mobile applications would improve the situation significantly and allow regression testing also on mobile devices.

The idea in automated GUI testing is to develop testing applications where the programmer defines the interaction points between the user and the GUI application that she wants to test. The test simulates user behavior and GUI responses; it defines and executes a particular scenario to discover possible deviations from the expected behavior. The tests are structured against the GUI application to elucidate and clarify what is required from user perspective.

To demonstrate the two testing approaches, we use a simple contact GUI application which allows to store and view a person's contacts with its addresses. The application (see Fig. 1) consists of one activity, which contains several edit fields to enter the person's data, buttons to save, edit, and delete data, and a more complex table control to list and view the stored contacts.

Android Instrumentation Framework

The *Android Instrumentation Framework* is integrated in the Android software development kit (sdk). It is located in the *android.test* package. Instrumentation refers to the ability to monitor and diagnose an application by inserting tracking code, debugging techniques, performance counters, and event logs into the code, which also allows measuring its performance and control its behavior [DM07].

1 The original version of this paper has been accepted at the International Conference on Testing Software an Systems. ICTSS2010, Natal, Brasil

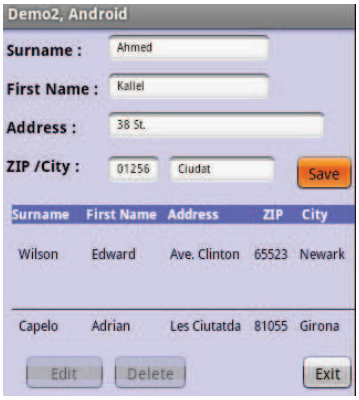

Figure 1: Contact GUI Application

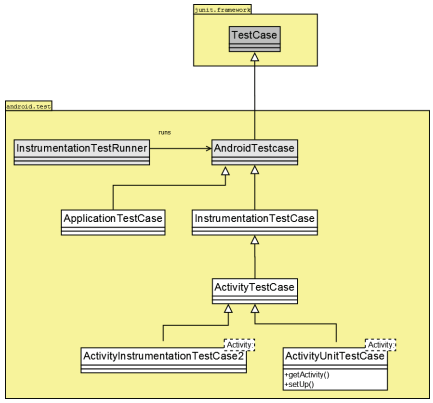

Figure 2: Android Instrumentation Framework Class Diagram

The framework includes test supporting classes (see Fig. 2), which bring an instance of the app under test to the test stage, start, manage and terminate this instance in test mode. Also, depending on the testing level [ADR] those classes provide facilities for controlling and ensuring the access through the application and its resources [AIF]. The framework is based on the *JUnit* framework by extending the *JUnit* core *TestCase* class. This allows using the standard JUnit assert-functionality for verifying expected and actual behavior in the GUI related to user's interactions, fired events, etc. [DM07]. This makes using the instrumentation framework, at least for experienced *JUnit* developers, very easy.

An Android application typically consists of many screens to interact with the user; each screen is represented by an Android activity, so an Android application can be seen an activity stack. So the Android GUI is composed of activities which in themselves are groups of ordered UI elements and where each activity has its independent lifecycle. Hence, each activity can be tested separately. Android instrumentation provides the special class *ActivityInstrumentationTestCase2* for this testing level. It offers more options for configuring the activity under test than its now deprecated predecessor *ActivityInstrumentationTestCase*. *ActivityInstrumentationTestCase2* provides functionality for handling resources and accessing the GUI on the activity context level. The programmer creates a test class for each activity and specifies the activity class to be tested (class under test – CUT).

An instrumentation GUI test should simulate a process related to the UI elements in the activity or a specific expected user behavior [ADR]. The general structure for an instrumentation test class is shown in Listing 1. This sample demonstrates a test for adding a new person in the demo application. The low level API of the instrumentation framework allows sending key strokes to the selected control using the *sendkey()* method for simulating specific user interactions; in the sample to enter the surname of a person.

```
public TestDemo(String pkg, Class<Demo>activityClass) {
  public TestDemo(){
  super("org.demo", Demo.class);//Bundle activity
}
protected void setUp() throws Exception {
  super.setUp();
  final Demo2 a = getActivity();//Instance Activity
  /*Bundle widgets by id in file R*/
  surname = (TextView)a.findViewById(R.id.surname);
}
public void test1AddPerson (){
  sendKeys( KeyEvent.KEYCODE_T); //Type a letter
  //runnable for save button action event
  Runnable saveRun = new Runnable() {
    public void run() {
      save.performClick();//Simulate click event
      p = getActivity().getPerson();//get data from UI

      save.setEnabled(false);//set UI element status
      Assert.assertFalse(save.isEnabled());
    }
  };
  getActivity().runOnUiThread(saveRun);//Run test
}
```

Listing 1: Test Class Structure with Android Instrumentation Framework.

For testing user action events (e.g. to click the 'Save' button in Listing 1) the instrumentation framework is executed as a *Runnable*. The *Runnable* instance is passed to the UI thread which runs in parallel with the tested activity. The *Runnable* class allows to modify the *run()* method to include assertions and check them in the UI thread at the moment the event is fired.

The instrumentation framework uses the *JUnit* assertions to verify the GUI state and behavior. No new verification concepts are introduced, which makes it easy to use for experienced *JUnit* tester. The tests are compiled and bundled as an independent instance of the application.

Positron Framework

The *Positron Framework* is a client-server model built on top of the Android instrumentation framework to handle the activity's resources and offering a *Selenium*-like high-level approach for running test cases [APF]. The framework provides the communication network and the server services in the client-server model [APF]. Each test case is treated as a client, which connects to the server component that runs the activity [Sel]. The communication network services use the capabilities of *adb*'s, the *Android Debug Bridge*, provided by Android IDE, to establish the connection between the server and the clients [APF]. Each test method (client) creates its own activity instance to communicate with.

The Positron framework is thread-safe, which is especially important to control the UI elements and their events running in a separate thread [APF]. To access activity resources it uses a path with dot-separated property notation [Sel]. It considers the activity as a container of UI elements arranged in a hierarchy. In addition, Positron provides variations of the method *at()* to tweak the return type according to the required property, such as: *stringAt()*, *intAt()*, for example.

The user written test class must extend class *TestCase*. The structure of the test class is similar to the structure of the standard *JUnit* test class. Also, all verifications and test about behaviors and data are made by *asserts*, which are structured as in *JUnit* [Sel]. In Listing 2, we show a typical test class written with Positron and some important methods to simulate user interactions with the GUI.

Positron includes its own Selenium like implementation to allow automated run of high-level GUI test suites, which contains a test class for each Selenium story. Each test class consists of a set of test methods representing a user tasks. This enables to run the tests independently of each other [ADG]. In addition, Positron establishes provides synchronization between the app under test and the test suite about the use of the resources. This allows the tests classes to use the needed resources from the activity and, at the same time, ensures the correct function of methods, widgets, etc., in the testing scenarios. The negotiation is made by Positron when it opens the connection between server and client, i.e. when the activity is started in test mode [APF]. This architecture allows a more development cycle and an easy IDE integration.

Framework Comparison

In contrast to desktop development, in mobile development there are not many options available for GUI testing. There are no general purpose test frameworks yet, so the presented frameworks are specific to the Android environment.

Comparing both approaches, the Android Instrumentation Framework, by bringing the activity into the test class, gives a handle to the context used to poke freely around the activity to validate test assertions, and to access the properties of the widgets directly in the test case, which means efficient runtime and fast answering. The Positron Framework connects to the application under test each time that the test class needs to use activity's resources, which means slows down communication to the activity under test.

With Android Instrumentation Framework, every UI element must be brought individually

```
Case {
  @Before
  public void runBeforeEveryTest() {
    //Start the activity in test mode
    startActivity("org.demo.Demo","org.demo.demo2.Demo2");
    pause();//Wait for the requested answer
    press("Name", DOWN); //Simulate typing a word
    click();//Simulate click event in a focused UI element
  }
  @Test
  public void addPerson() throws InterruptedException {
    /*Assert state*/
    assertTrue("not click", booleanAt("save.isPressed"));
    field1 = stringAt("listView.1.0.text");//Get string
  }
}
```

Listing 2: Test Class Structure with Positron.

```
//ANDROID INSTRUMENTATION FRAMEWORK
   sendKeys( KeyEvent.KEYCODE_N);
   sendKeys( KeyEvent.KEYCODE_A);
   sendKeys( KeyEvent.KEYCODE_M);
   sendKeys( KeyEvent.KEYCODE_E);
//POSITRON FRAMEWORK:
   press("NAME);
```

Listing 3: Simulate Typing action with both Approaches

in the setup method; i.e., the GUI is simulated as in the activity under test (see in Listing 1, *setup()* method). Also, the methods for handling user action events must be overridden as *runnable*. Although, this allows writing test cases to test GUI at a much lower level than UI screen shot tests, the tester need to write many lines code, which makes test writing error prone, itself.

Instead, Positron provides the implementation of methods to locate activity resources by calling getters for each named property class. This avoids a rewriting of the activity within the test class (see in Listing 2, *setup()* method). On the application code side, this requires that the activity's class under test has implemented the corresponding getters methods to the resources. The UI objects' methods are handled by Positron. The developers do not need to take core of the synchronization between the UI thread and UI objects methods. However, this limits the UI Objects handling at screen level.

For writing tests with Android Instrumentation Framework, the developers use low-level methods to simulate user interactions with the screen; while Positron provides specific high-level methods for the simulation of events and user processes. Listing 3 shows for each framework the statements, to simulate entering the word 'name' into an edit field.

In summary the Android instrumentation framework offers greatest flexibility and direct access to the GUI controls through its low-level API. This however requires the user to write more test code, which increases chances for new errors and also causes higher maintenance effort. The Positron framework on the other side provides a high-level interface for writing automated GUI tests, which reduces the effort, for both, writing and maintaining test code significantly.

The notable strengths of both frameworks are: the use of instrumentations for handling UI resources through the activity; user interactions are simulated by sending key events; tests are run on target platform; assertions allow verification of particular features or behavior in the GUI and the states of UI elements.

Among the weaknesses of these approaches are that the tester must have a detailed knowledge of the source code under test to find the UI resources in the code. The developers cannot write tests spanning multiple activities, because the frameworks isolate the test class for handling only one activity from an activity stack.

Conclusion and Outlook

Our analysis of GUI testing tools for mobile application development, exemplified by the Android platform as one of the currently most emerging environments, showed, that this is area is still in its beginning. The two analyzed frameworks, the instrumentation framework and the Positron framework, provide basic GUI testing functionality on various levels. Compared to GUI desktop testing tools, like Selenium HQ [Sel], for example, which offers more capabilities to reach better performance in writing automated GUI tests, both frameworks yet show notable limitations.

These limitations include: No capture/replay functionality, limited support for GUI controls, script generation, and limited runtime control in a compulsory interruption environment. Other limitations, that are specific to mobile application development are, platform neutral testing, i.e. testing of the various OS and runtime platform of the very defragmented device market, or even language neutral testing platforms.

In a current research project at our institute we are developing a testing approach to verify that a ported mobile application is following a given architecture [MobPal].

References

[ADG] Android Developers, Dev Guide, Testing and Instrumentation. http://developer.android.com/guide/topics/testing/testing_android.html, 18.07.2010

[ADR] Android Developers, Reference, Android Test. http://developer.android.com/reference/android/test/package-summary.html, 18.07.2010

[AIF] Android Platform Development Kit, Instrumentation Framework. Google, 2008. http://www.netmite.com/android/mydroid/development/pdk/docs/instrumentation_framework.html, 18.07.2010

[APF] Autoandroid. Positron Framework. http://code.google.com/p/autoandroid/wiki/Positron

[DM07] Dalle, O., Mrabet, C. An Instrumentation Framework for component-based simulations based on the Separation of Concerns paradigm. Proc. of 6th EUROSIM Congress, 2007.

[Ger97] Gerrard, P. Testing GUI Applications. EuroSTAR Conference, Edinburgh, November 1997. http://www.gerrardconsulting.com/GUI/TestGui.html, 18.07.2010

[Mar98] Marick, B. When Should a Test Be Automated? Presented at Quality Week, 1998. http://www.exampler.com

[MobPal] Research Project: Mobile Paladin. CTI Project 10829.1;3 PFES-ES, 1.12.2009-1.12.2010.

[Sel] Selenium. Selenium-IDE. http://seleniumhq.org/docs/03_selenium_ide.html, 18.07.2010