Testing on Android

Mobile Application Development

MTAT.03.262 2016 Fall

Jakob Mass
jakob.mass@ut.ee
Introduction.

- Perfect codewriting...or?
- Conventional (unit) Java testing with JUnit
- How is mobile/Android testing special?
- Continuous Integration
Typical Java Unit testing

JUnit framework offers convenient ways to perform reusable setup, teardown and assertion operations.

@Test - annotation

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void evaluatesExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        assertEquals(6, sum);
    }
}
```

https://developer.android.com/training/testing/start/index.html#junit
Part 1. Android Unit tests

Fundamental tests in a software testing strategy

- Test smallest possible units of code
- Isolation
  - use mocking frameworks to isolate your unit from its dependencies

In Android, 2 types of unit tests:

1. Local Unit tests
   a. Run on local JVM
   b. No access to Android framework API
      i. You could mock some of the API

2. Instrumented tests
   a. Run on a physical Android device or emulator

https://developer.android.com/training/testing/start/index.html#test-types
- JVM execution -- **fast!**
- Can’t use Android framework dependencies
  - Possible to use **mock** objects instead
- Access to instrumentation info (e.g. app **Context**)
- Use if you can’t easily mock your Android framework objects

https://developer.android.com/training/testing/start/index.html#test-types
Local Unit Tests

Demo - Simple Unit test using Android Studio

Set-up (done by default in Android Studio):

- Test files in module-name/src/test/java/
- JUnit 4 dependency in your app's build.gradle

```gradle
dependencies {
    ...
    testCompile 'junit:junit:4.12'
}
```

Let’s write our first unit test!

- Shopping List App

https://developer.android.com/training/testing/unit-testing/local-unit-tests.html
Unit test isolation discussion

- Unit tests are executed using a modified android.jar library, which is empty!
- This helps encourage the “test-in-isolation” mindset

So any time you find your test not working because of this, you have 2 options:

1) Rethink your tests, are you really testing independent units?

2) Isolate through using mock objects
Local Unit Testing: Mocking with Mockito

Let’s add another dependency:

● dependencies {
  ...
  testCompile 'org.mockito:mockito-core:1.10.19'
}

● @RunWith(MockitoJUnitRunner.class) annotation

Creating a mock Android object:
● add the @Mock annotation before the field declaration.
● Define the behaviour:
  when( X ).thenReturn( Y ) methods
● Matchers: anyString(), eq(), ....

https://developer.android.com/training/testing/unit-testing/local-unit-tests.html#mocking-dependencies
Moving on to Instrumented tests..

https://developer.android.com/training/testing/start/index.html
Android Testing Support Library

a framework that allow to help you build and run test code for your apps

Includes:

- AndroidJUnitRunner
- Espresso
- UI Automator

ATSL is included with the Android Support Repository which you can obtain using the SDK manager

https://google.github.io/android-testing-support-library/
Instrumented Unit Tests

Set-up:

- Make sure you’ve got **Android Support Repository** installed in your SDK manager!
- Dependencies in your app’s `build.gradle`:
  ```groovy
  dependencies {
      ...
      androidTestCompile 'com.android.support:support-annotations:24.2.1'
      androidTestCompile 'com.android.support.test:runner:0.5'
      androidTestCompile 'com.android.support.test:rules:0.5'
  }
  ```

- Set AndroidJUnitRunner as the default test instrumentation runner:
  ```groovy
  defaultConfig {
      ...
      testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
  }
  ```
Test suites, Firebase Test Lab

- Instrumented tests can be grouped together to form test suites

```java
import com.example.android.testing.mysample.CalculatorAddParameterizedTest;
import com.example.android.testing.mysample.CalculatorInstrumentationTest;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

// Runs all unit tests.
@RunWith(Suite.class)
@Suite.SuiteClasses({
    CalculatorInstrumentationTest.class,
    CalculatorAddParameterizedTest.class
})
public class UnitTestSuite {}```

- Additionally, Android Studio also provides means for deploying tests to Google’s **Firebase Test Lab**

- **Firebase Test Lab** runs your tests on physical configurations
Automated UI testing

Unit tests aren’t suitable for testing complex UI interaction events.

This type of testing takes an user/view-centric perspective to your application

Generally, UI testing in Android can be divided into 2:

- **UI tests that span a single app**
  - Typically to check that UI outputs to user interactions are correct
- **UI tests that span multiple apps**
  - Verifying the behaviour that takes places when different apps (system and non-system) interact.
Single App UI testing with Espresso

● Espresso Framework
  ○ Programmatically simulate user interactions (click, swipe, text input, ..)
  ○ run on Android 2.3.3 (API level 10) and up

● Espresso has focused on solving one of the bigger challenges in automated UI testing on Android: synchronizing actions
  ○ Earlier, you’d often add Thread.sleep()-s in your tests to make sure the execution has followed up with your test

● dependencies {
  ...
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'
    // Optional -- Hamcrest library
    androidTestCompile 'org.hamcrest:hamcrest-library:1.3'
  }

● Android Developer docs suggest turning off animations on your test device:
  ○ Developer options-> Window animation scale, Transition animation scale, Animator duration scale
Espresso Test Basics

1. Select a View you want to work with using
   a. `onView()` (for Activities)
   b. `onData()` (for AdapterViews)

2. Simulate specific interaction using
   a. `perform()`

3. Repeat previous steps to simulate user flow

4. Use ViewAssertion methods to verify if actual behaviour matches expected

```java
onView(withId(R.id.my_view))
    .perform(click())
    .check(matches(isDisplayed()));  // withId(R.id.my_view) is a ViewMatcher
```

https://developer.android.com/training/testing/ui-testing/espresso-testing.html
Espresso test example with ActivityTestRule

Demo...

The important idea is to use the following to reduce boilerplate code

```java
@Rule
public ActivityTestRule<MyActivity> mActivityRule =
    new ActivityTestRule<*>( MyActivity.class );
```
Advanced Android Espresso
by Chiu-Ki Chan

@chiuki
+ChiuKiChan

UI testing with multiple apps

- It’s not uncommon for user flow to span multiple apps
  - E.g. In HA1 our contact picker app opened an e-mail app, and returned to the original application

- This type of flow can be verified using the UI Automator APIs
UI Automator basic approach

1. Get the `UiDevice` object, this is reflects the device you are testing

2. Get the `UiObject` object, by calling `findObject()`

3. Simulate user interaction on the `UiObject`, similar to Espresso

4. Check that the behaviour matches the expected one

```
// Initialize UiDevice instance
mDevice = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());

UiObject okButton = mDevice.findObject(new UiSelector()
    .text("OK")
    .className("android.widget.Button"));

// Simulate a user-click on the OK button, if found.
if(okButton.exists() && okButton.isEnabled()) {
    okButton.click();
}
```
Continuous Integration

Now that you have your tests, you can create a nice development-deployment flow using CI

For example:

1. You update your application code
2. Push to VCS
3. CI sees the new changes, builds and tests
4. Based on the results
   a. Failure: Notify the developers e.g. via E-mail, Slack
   b. Success: Deploy the app e.g. to the Play Store

Greenhouse CI

Tartu-based company, support for iOS and Android

https://greenhouseci.com/
Thanks for listening!