Testing Android Applications

Mobile Computing & Internet of Things
LTAT.06.009

Jakob Mass
jakob.mass at ut.ee
Introduction

• How to write perfect code?

• Are you familiar with unit testing in Java / Javascript / .NET / ... ?
Testing - Fundamentals

• Separate responsibilities into modular units
  • Iterative process, you may have to (re-)think about responsibilities with every new feature!

• For each unit, write a set of unit tests
  • Should be exhaustive
    • Consider all interactions
    • Corner cases
    • Consider possible inputs (incl. invalid ones!)

• Validate integration between units with UI tests & Integration tests
  • Flows & interactions which encompass several units

https://developer.android.com/training/testing/fundamentals
Testing Pyramid

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Validates interactions...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Tests</td>
<td>..of a single unit, one class at a time</td>
</tr>
<tr>
<td>(Unit tests)</td>
<td></td>
</tr>
<tr>
<td>Medium Tests</td>
<td>a) ..between levels of stack within a module</td>
</tr>
<tr>
<td>(Integration tests)</td>
<td>b) ..between related modules</td>
</tr>
<tr>
<td>Large Tests</td>
<td>..of user journeys through multiple app modules</td>
</tr>
<tr>
<td>(UI tests)</td>
<td></td>
</tr>
</tbody>
</table>

https://developer.android.com/training/testing/fundamentals
Unit tests

*Fundamental tests in a software testing strategy*

- Test smallest possible units of code
  - Class, method, component
- Exhaustively validate the functionality and contracts of each class within your app.
- Units usually tested in isolation
  - Affect and monitor changes only to the unit under test
  - Sometimes, isolating an unit is not straight-forward:
    - Mock the behaviour of the components which unit depends on

https://developer.android.com/training/testing/unit-testing/
Typical Java Unit testing

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

```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
Android Unit testing

There are two types:

1. **Local Unit tests**
   - Run on local JVM
   - Isolate unit from dependencies using mock objects
   - No access to Android framework API
     - Can use a framework like Roboelectric, a dependency provider

2. **Instrumented tests**
   - Run on an Android device (physical/emulator)

[https://developer.android.com/training/testing/unit-testing](https://developer.android.com/training/testing/unit-testing)
• JVM execution – **fast**!

• Can’t use Android framework dependencies
  • Can use mock objects instead

• Access to instrumentation info (e.g. app **Context**)

• Use if you can’t easily isolate the framework dependencies
Local Unit Tests

Set-up:
- Test files in module-name/src/test/java
- JUnit 4 dependency in your app’s build.gradle
  - *(done by default in Android Studio)*

```
// Required -- JUnit 4 framework
testImplementation 'junit:junit:4.12'

// Optional -- Robolectric environment
testImplementation 'androidx.test:core:1.2.0'
// Optional -- Mockito framework
testImplementation 'org.mockito:mockito-core:1.10.19'
```

Let’s run our first unit test!

https://developer.android.com/training/testing/unit-testing/local-unit-tests.html
Test Subject: Shopping Basket App

2 classes:
- **ShoppingItem**
  - Name, price, quantity
  - Price determined by no. of letters!
- **ShoppingBasket**:
  - Holds items in a collection
  - Can add items
    - re-adding item with same name updates items quantity
  - Calculates total price of items in basket

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Egg</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Milk</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Total Cost: $15
Demo

class ExampleUnitTest {
    private lateinit var basket: ShoppingBasket

    // Note: "@Before" gets called before each @Test method
    @Before
    fun initEmptyBasket() {
        basket = ShoppingBasket()
    }

    @Test
    fun itemPrice() {
        val item = ShoppingItem("Milk")
        assertEquals(4, item.price)
    }
}
```kotlin
@Test
fun problematicUnitTest() {
    val item3 = ShoppingItem("12345")
    // getReformattedName() uses some Android library functions
    item3.getReformattedName()
    assertEquals("Product 12345", item3.name)
}
```

Tests failed: 1 of 1 test - 11 ms

```
    at android.text.TextUtils.isDigitsOnly(TextUtils.java)
    at ee.ut.cs.shoppingbasket.ShoppingItem
        .getReformattedName(ShoppingItem.kt:12)
    at ee.ut.cs.shoppingbasket.ExampleUnitTest/
        .problematicUnitTest(ExampleUnitTest.kt:84)
```
Unit test isolation discussion

- Unit tests are executed using a modified android.jar library, which has empty implementations
- Helps encourage the “test-in-isolation” mindset

So any time you find your test not working because of this, you can either:

1) Rethink your tests- are you really testing independent units?
2) Isolate by using mock objects
3) Use a framework like Roboelectric*

* [http://robolectric.org/](http://robolectric.org/)
Local Unit Testing: Mocking with Mockito

Annotate your test class with:

```
@RunWith(MockitoJUnitRunner::class)
```

Creating a mock Android object:
- Add the `@Mock` annotation to 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
@RunWith(MockitoJUnitRunner::class)
class UnitTestSample {

  @Mock
  private lateinit var mockContext: Context

  @Test
  fun readStringFromContext_LocalizedString() {
    // Given a mocked Context injected into the object under test...
    `when`(mockContext.getString(R.string.hello_word)).thenReturn(FAKE_STRING)
    val myObjectUnderTest = ClassUnderTest(mockContext)

    // ...when the string is returned from the object under test...
    val result: String = myObjectUnderTest.getHelloWorldString()

    // ...then the result should be the expected one.
    assertEquals(result, `is`(FAKE_STRING))
  }
}
Roboelectric

Simulates the runtime for Android API level 16+
  • provides community-maintained fakes called *shadows*.

• Can run on a JVM-powered development machine
  • This allows you to test code that depends on the framework without using mock objects or an emulator.

• Supports the following aspects of the Android platform:
  • Component lifecycles
  • Event loops
  • All resources

http://robolectric.org/
Code coverage

• Code coverage reports show metrics how much of the source code has been reached via automated test execution
  • Different criteria: covered no. of functions, classes, lines, ...

Being proud of 100% test coverage is like being proud of reading every word in the newspaper. Some are more important than others.

7:55 PM - Dec 24, 2016
59 replies 2,704 retweets 3,352 likes
Moving on to Instrumented tests...

https://developer.android.com/training/testing/start/index.html#test-types
AndroidX Test libraries & APIs

A set of libraries for testing APIs, including instrumentation and Roboelectric support

Includes:
• AndroidJUnitRunner
• Espresso
• UI Automator

*Part of Android Jetpack*

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

• Dependencies:

```java
androidTestImplementation 'androidx.test:runner:1.2.0'
adventTestImplementation 'androidx.test:rules:1.2.0'
```

• Set AndroidJUnitRunner as the default test instrumentation runner:

```java
defaultConfig {
    ...
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}
```

https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests#setup
Instrumented unit test class

• Similar to the local JUnit 4 test class
  • @RunWith(AndroidJUnit4::class)

• Instrumented tests can be grouped together to form test suites
  • @RunWith(Suite::class), @Suite.SuitClasses() annotations

Example:
Test suite with name UnitTestSuite
Groups 2 classes:
1. CalculatorInstrumentationTest
2. CalculatorAddParameterizedTest

https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests#test-suites
Firebase Test Lab

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

Firebase Test Lab runs your tests on virtual and physical devices with various configurations.

https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests#run-ctl
Demo

```kotlin
@Test
fun useAppContext() {
    // Context of the app under test.
    val appContext = InstrumentationRegistry.getInstrumentation().targetContext
    assertEquals("ee.ut.cs.shoppingbasket", appContext.packageName)
}
```
Single-App UI testing

Using Espresso
UI testing with Espresso Framework

• Programmatically simulates user interactions
  • Click, swipe, text input, ...
  • Run on Android 2.3.3 (API v10) and up
• Espresso manages synchronization of UI actions
  • Can be a huge challenge in automated UI testing
Setting up Espresso

• Dependencies:

```java
androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
```

• Android Developer docs advise turning off animations on test device
Espresso Basics

1. Select a View to work with:
   a) `onView(targetView)` (for Activities)
   b) `onData()` (for AdapterViews)

2. Simulate specific interaction:
   a) `perform(action)`

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

https://developer.android.com/training/testing/ui-testing/espresso-testing.html#build
https://developer.android.com/training/testing/espresso/basics
assertEquals(stupidSimpleMockObject.toDouble(), 42.0)

assertThat(stupidSimpleMockObject.toDouble(), `is`(42.0))

- Espresso is most often used together with Hamcrest matchers
- Hamcrest is a library for declarative style matcher predicates – very useful for testing

// Instead of:
var found = false
for (pet : Pet in petList) {
    if (pet == dog) found = true
}
assertEquals(true, found)

// with hamcrest matchers:
assertThat(petList, hasItem(dog))

https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki
Espresso ActivityTestRule

• JUnit rules:
  • mechanism to enhance tests by running code around a test case execution

• AndroidX test includes some useful JUnit rules to reduce boilerplate code:

  • **ActivityTestRule**
    • Launches and closes specified activity before each test case

```
@get:Rule
var activityRule: ActivityTestRule<MainActivity> = ActivityTestRule(MainActivity::class.java)
```

• **ServiceTestRule**

https://developer.android.com/training/testing/junit-rules
Espresso Intents

An extension library to Espresso

- Validate and stub (mock) intents sent out by the app
- Intercept outgoing intents, stub the result, send back the result to the component under test
- focus on your own app's logic while assuming that other apps or the platform will function correctly

`androidTestImplementation 'androidx.test.espresso:espresso-intents:3.1.0'

- Initialized in test cases via IntentsTestRule

```java
@get:Rule
var rule: IntentsTestRule<MyActivity> = IntentsTestRule(MyActivity::class.java)
```

https://developer.android.com/training/testing/espresso/intents
Espresso Intents example

```java
@RunWith(AndroidJUnit4::class)
class SimpleIntentTest {

    @get:Rule
    var intentsRule: IntentsTestRule<MyActivity> = IntentsTestRule(MyActivity::class.java)

    @Test
    fun verifyMessageSentToDisplayActivity() {
        // Type a message into a EditText
        onView(withId(R.id.editText_name))
            .perform(typeText(MESSAGE), closeSoftKeyboard())

        // Button should send message to another activity via an explicit intent.
        onView(withId(R.id.button)).perform(click())

        // Verify that DisplayMessageActivity got intent
        // with correct package name and message
        intended(allOf(
            hasComponent(hasShortClassName(".DisplayMessageActivity")),
            toPackage(PACKAGE_NAME),
            hasExtra("EXTRA_KEY", MESSAGE)))
    }
```
UI testing with multiple apps

• It’s not uncommon for user flow to span multiple apps
  • Remember HW4? The app opened the telephone dialer, and then returned to the original application

• This type of flow can be handled using the **UI Automator** APIs
  • Perform interactions on user and system apps
  • E.g. opening menus and app launchers

https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html
UI Automator basic approach

1. Get the **UiDevice** object, this reflects the device being tested
2. Get the **UiObject** object, by calling **findObject()**
3. Simulate user interaction on UiObject, similar to Espresso
   • click(), dragTo(), setText(), clearTextField(), swipeUp(), ..
4. Check that the behaviour matches expected one with Junit asserts

Can also launch intents directly

https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html
UI Automator code sample

```java
// Initialize UiDevice instance
device = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation())

val cancelButton: UiObject = device.findObject(
    UiSelector().text("Cancel").className("android.widget.Button")
)

val okButton: UiObject = device.findObject(
    UiSelector().text("OK").className("android.widget.Button")
)

// Simulate a user-click on the OK button, if found.
if (okButton.exists() && okButton.isEnabled) {
    okButton.click()
}
```
Our skills so far..

- Core Topics:
  - App. Lifecycle
  - Components
    - Intents, Services, Broadcasts, Content Providers

- Handling Data:
  - User state, Storage, File I/O, Resources

- Constructing UIs:
  - ConstraintLayout, Fragments
  - Bridging Data and UI: Adapters, ViewModel

- Background tasks, Testing

Home Assignment 1