MTAT.03.262
Mobile Application Development
Testing on Android

Jakob Mass
jakob.mass@ut.ee
Introduction

• How to write perfect code?

• Familiar with Unit testing in Java?
Overview

- Conventional Java testing
- What makes mobile/Android testing special?
- Further concepts: Continuous Integration
Unit tests

• Fundamental tests in a software testing strategy
  • Test smallest possible units of code
  • Isolation*
    • Use mocking framework to isolate an unit from its dependencies

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
   • No access to Android framework API
     • You could *mock* some of the API

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

https://developer.android.com/training/testing/start/index.html#test-types
- 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 mock the framework objects

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

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`

```groovy
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

Let’s write our first unit test!

Demo – Simple Unit test using Android Studio

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

- ShoppingItem
  - Name, price, quantity
  - Price determined by no. of letters!

- ShoppingBasket:
  - Adding items
  - Item names are unique, re-adding same item updates items quantity
  - Calculating total price of items in basket
• Course Exercise:
  • Simple Unit tests (Follow lecturer example)
Unit test isolation discussion

• Unit tests are executed using a modified android.jar library, which is empty!
• 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 something like Roboelectric*

* http://robolectric.org/
Local Unit Testing: Mocking with Mockito

Add dependency:
```java
testImplementation 'org.mockito:mockito-core:1.10.19'
```

Annotate your test class with:
```java
@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
Code Coverage

• See if you can get 100% coverage for ShoppingBasket class

COURSE EXERCISE:
GENERATE A HTML CODE COVERAGE REPORT
(INCLUDE IN YOUR PROJECT SUBMISSION .ZIP)
Moving on to Instrumented tests...

https://developer.android.com/training/testing/start/index.html#test-types
Android Testing Support Library (ATSL)

Framework that helps building and running app test code

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

• Make sure you have Android Support Repository installed in SDK Manager!

• Dependencies:

```java
androidTestImplementation 'com.android.support:support-annotations:28.0.0'
androidTestImplementation 'com.android.support.test:runner:1.0.2'
androidTestImplementation 'com.android.support.test:rules:1.0.2'
```

• Set AndroidJUnitRunner as the default test instrumentation runner:

```java
defaultConfig {
  ...
  testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
}
```

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

- Instrumented tests can be grouped together to form **test suites**

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

Firebase Test Lab runs your tests on physical devices with various configurations
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 { }
Single App UI testing with Espresso Framework

- Programmaticaly 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:

androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
// Optional -- Hamcrest library
androidTestImplementation 'org.hamcrest:hamcrest-library:1.3'

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

```
onView(withId(R.id.my_view))
    .perform(click())
    .check(matches(isDisplayed()));
```

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

• Demo

Important: use the following to reduce boilerplate code:

```java
@Rule
public ActivityTestRule<MyActivity> mActivityRule =
    new ActivityTestRule<>(MyActivity.class);
```
Further Espresso concepts

E.g. Espresso Intents:

- 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
UI testing with multiple apps

• It’s not uncommon for user flow to span multiple apps
  • Remember HA1? The contacts app opened the e-mail app, and then returned to the original application

• This type of flow can be handled using the UI Automator APIs

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
4. Check that the behaviour matches 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 app + tests, you can start thinking about automating other aspects of your software project

Continuous Integration

- Practice of merging code changes across developers, as frequently as possible

- Validate changes with automated builds & tests
  - Includes static code analysis, testing and configuration sets

- Can include Continuous Deployment:
  - Making the newest version available somewhere
Continuous Integration

1. You update your application code
2. Push to VCS (such as GitHub)
3. CI system sees the new changes, pulls the latest code, builds and runs tests
   - Perhaps on different target API levels
4. Based on build & test result:
   - Failure: Notify developers via E-mail, Slack, ...
   - Success: Deploy to Play Store, to Test Audience, Staging Environment, etc

- CircleCi,
- NeverCode - company with roots in Tartu, support for iOS and Android
Summary

• Local Tests vs Instrumented Tests
• Unit Testing
• UI testing
  • Espresso
  • UIAutomator
• Testing philosophy
  • Isolation of components
  • Mocking objects
• Additionally:
  • Firebase Test Lab
  • Continuous Integration