DA: Android Basic Testing

Applications for mobile devices - Theory - Unit 3

Didac Florensa Cazorla

Any: 2021-2022
Curs: 102386
Institut: University of Lleida (Campus Igualada)
Titulació: Bachelor's degree in Digital Interaction and Computing Techniques (GTIDIC)

Agenda

Introduction

Introduction to testing

Unit Testing

Integration Testing

Espresso

Mocking

Introduction

Warm-up

- Dr. Jordi Mateo Fornés
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: jordi.mateo@udl.cat
- **Twitter**: https://twitter.com/MatForJordi
- **Github**: https://github/JordiMateoUdL

- Ph.D Dídac Florensa Cazorla
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: didac.florensa@udl.cat

Introduction to testing

Why do we need to test

- Measure the impact of change in code.
- Rapid feedback on failures.
- Early failure detection in the development cycle.
- Safer code refactoring, letting you optimize code without worrying about regressions.
- Stable development velocity, helping you minimize technical debt.

Test Automation: Objective

- A code that **automatically** check that **right input** produces **right output**.
- A code that **automatically** does all the things that the **user would do**?.
  - Fill in the correct values on each screen.
  - Click all the buttons.
  - ...
- We want to eliminate the manual testing overhead.

Test Automation: Why

- It is not faster to test manual a slight change on my code and get the feedback. Why do I need to automate this test?
- The first time is cheaper to do it manually.
  - What happens if, in the future, I change things?
  - What happens if I run 10 times to ensure it works properly.
- We need to **MAINTAIN** the app, and we can not predict the impact of our future changes.

Test Coverage

- Metrics that estimate how our test covers our application.
- It is not a guarantee is just an estimation.
- There are a lot of metrics the developers can use to do this coverage.
- It is not a measurement of how good the software is.

Types of test

- **Unit test** -> Isolating components
- **Integration test (Instrumentation test**) -> Testing components in their context with other components interaction.
- **User Acceptance Testing** -> How the final user is going to use it.

Testing Pyramid

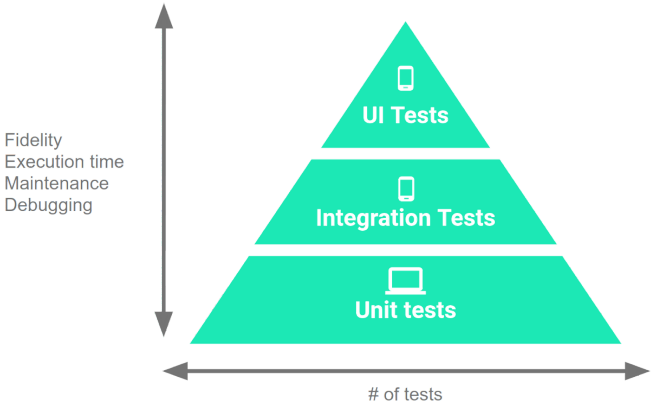

Figure 1: The Testing Pyramid, showing the three categories of tests that you should include in your app's test suite https://developer.android.com/

Unit Testing

What is Unit Testing

- Is a method to validate that each code unit performs as expected.
- A unit can be an individual function, method, procedure, module or entire object.
- This method must be applied during the developing step.
- *Junit*: a free to use testing tool used for Java programming language. This tool tests data first and then inserts it into the piece of code. It provides assertions to identify the test method.

Unit Testing: Pros and Cons

Advantages

- Easily verify that the logic of individual units is correct.
- Allows safe refactoring.
- We can use a mock object to test parts that are not completed.

Disadvantages

- We can not evaluate all possible scenarios.
- We can not catch integration errors.

Hands-On (**UnitTest**): *Login Activity*

- **TODO**: We are going to create a simple login activity and add unit tests.
- **TODO**: LoginUtils.class:
  - **TODO**: isValidEmailAddress(String email).
  - **TODO**: isValidPassword(String password).

Hands-On (**UnitTest**): *LoginUtils.class*

```
public boolean isValidEmailAddress(String email){
        final String MAIL_PATTERN =
        "^[\\w-_.+]*[\\w-_.]@([\\w]+\\.)+[\\w]+[\\w]$";
        return email.matches(MAIL_PATTERN);
    }
public boolean isValidPassword(final String password) {
        Pattern pattern; Matcher matcher;
        final String PASSWORD_PATTERN =
        "^(?=.*[0-9])(?=.*[A-Z])(?=.*[@#$%^&+=!])(?=\\S+$).{8,}$";
        pattern = Pattern.compile(PASSWORD_PATTERN);
        matcher = pattern.matcher(password);
        return matcher.matches();
    }
```

Hands-On (**UnitTest**): *LoginUtilsTest.class* - Assert True

- Create a simple test to check if the function isValidEmailAddress works with a valid string.
- @Test -> All the tests must begin with this annotation.

```
@Test
 public void validAddressPasses() throws Exception {
   LoginUtils loginUtils = new LoginUtils();
   assertTrue(loginUtils.isValidEmailAddress("xxx@hotmail.com"));
   }
```

Hands-On (**UnitTest**): *LoginUtilsTest.class* - Assert False

- Create a simple test to check if the function isValidEmailAddress fails with an invalid string.

```
@Test
 public void invalidAddressFails() throws Exception {
   LoginUtils loginUtils = new LoginUtils();
   assertFalse(loginUtils.isValidEmailAddress("xxx"));
   }
```

Hands-On (UnitTest): *LoginUtilsTest.class* - @Before

- Note this line: LoginUtils loginUtils = new LoginUtils() is being duplicated, so **refacator**;
- @Before -> We are going to execute this function before each test.

```
private LoginUtils loginUtils;
@Before
   public void setUp(){
       loginUtils = new LoginUtils();
   }
@Test
 public void invalidAddressFails() throws Exception {
   assertFalse(loginUtils.isValidEmailAddress("xxx"));
 }
```

Hands-On (**UnitTest**): *LoginUtilsTest.class* - Why we throw an Exception

- Try to make a test that fails, for example:

```
@Test
 public void invalidAddressFails() throws Exception {
   assertTrue(loginUtils.isValidEmailAddress("xxx"));
 }
```

- It returns an exception. We can use this throws Exception to avoid the try/catch unless we want to test specific things with exceptions.

Integration Testing

What is an integration test (Instrumented)

- Instrumented test verify how different units collaborate.
- Simulate the same environment of the end-user.

You can run instrumented unit tests on a physical device or emulator. However, this form of testing involves significantly slower execution times than local unit tests, so it's best to rely on this method only when it's essential to evaluate your app's behaviour against actual device hardware.

Important tools

- Mockito.
- Espresso.
- Roboelectric.
- MockWebServer.

Hands-On **(Integration Testing)**: *GeoCoder*

- **TODO**: We are going to create a simple geocoder class.
- **TODO**: GeoCoderUtils.class:
  - **TODO**: getCountry(double lat, double lon).

Hands-On **(Integration Testing)**: *GeoCoderUtils*

```
public class GeoUtils {
  private Geocoder geocoder;
  public GeoUtils(Context context){
    geocoder = new Geocoder(context, Locale.getDefault());
  }
  public String getCountry(double lat, double lon)
  throws IOException {
    List<Address> addresses =
    geocoder.getFromLocation(lat, lon, 1);
    return addresses.get(0).getCountryName();
  }
}
```

Hands-On (**Integration Testing**): *GeoCoderUtils* - Dependencies

- Check your *build.gradle* (Most will be there by default!)

```
testImplementation 'junit:junit:4.13.2'
testImplementation 'org.mockito:mockito-core:2.21.0'
androidTestImplementation 'androidx.test.ext:junit:1.1.2'
androidTestImplementation
  'androidx.test.espresso:espresso-core:3.3.0'
androidTestImplementation 'com.android.support.test:rules:1.0.2'
androidTestImplementation
  'com.squareup.retrofit2:retrofit-mock:2.0.0'
androidTestImplementation
  ("com.squareup.okhttp3:mockwebserver:4.0.1")
```

Hands-On **(Integration Testing)**: *GeoCoderUtils* - Instrumentation Test

- Starts with the annotation **@RunWith(AndroidJUnit4.class)** to indicate that it is an instrumentation test and deploy it in a virtual device.

```
@Before
    public void setUp(){
      Context context =
      InstrumentationRegistry.getInstrumentation()
      .getTargetContext();
      geoUtils = new GeoUtils(context);
    }
```

Hands-On **(Integration Testing)**: *GeoCoderUtils* - Tests

```
@Test
  public void testSpainThatShouldPassWithIncorrectCoordinates()
  throws IOException {
    String countryName = geoUtils.getCountry(1.6172,41.58098);
    assertNotEquals("Spain", countryName);
  }

@Test
  public void testSpainThatShouldPassWithRightCoordinates()
  throws IOException {
    String countryName = geoUtils.getCountry(41.58098,1.6172);
    assertEquals("Spain", countryName);
  }
```

Hands-On **(Integration Testing)**: *GeoCoderUtils* - Handling with Exceptions

```
@Test(expected = IndexOutOfBoundsException.class)
  public void testShouldThrowIndexOutOfBoundsException()
  throws IOException {
    geoUtils.getCountry(0,0);
  }
```

Espresso

What is Espresso?

Google developed the framework to help you easily and quickly automate UI tests that simulate real user behaviour. These tests will run on a real device or emulator.

androidTestImplementation ‘androidx.test.espresso:espresso-core:3.3.0’

Espresso - Quick View

onView(), waits to perform the corresponding UI action or assertion until the following synchronization conditions are met:

- The message queue is empty.
- There are no instances of AsyncTask currently executing a task.
- All developer-defined idling resources are idle.

For more information, check this Testing espresso.

Hands-On **(UI Testing)**: *Login Activity* - View

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  login.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
      String pass = password.getText().toString();
      if (!loginUtils.isValidPassword(pass)){
        errorMsg.setText("Bad password");
        errorMsg.setVisibility(View.VISIBLE);
      }
    }
  });
}
```

Hands-On **(UI Testing)**: *Login Activity Instrumentation Test* - Set Up

```
@Rule
public ActivityTestRule<LoginActivity> mActivityRule
= new ActivityTestRule<>(LoginActivity.class);

@Before
public void setUp(){
  mActivityRule.getActivity();
}
```

Hands-On **(UI Testing)**: *Login Activity Instrumentation Test* - Test 1

```
@Test
public void testInvalidPasswordShowsErrorMsg(){
  Espresso.onView(ViewMatchers.withId(R.id.editText_pass))
    .perform(ViewActions.typeText("12345678"));
  Espresso.onView(ViewMatchers.withId(R.id.button_login))
    .perform(ViewActions.click());
  Espresso.onView(ViewMatchers.withId(R.id.errorMessage))
    .check(ViewAssertions.matches(ViewMatchers.isDisplayed()))
    .check(ViewAssertions.matches(ViewMatchers
      .withText("Bad password")));
}
```

Hands-On **(UI Testing)**: *Login Activity Instrumentation Test* - Test 2

```
@Test
public void testValidPasswordNotShowsErrorMsg(){
  Espresso.onView(ViewMatchers.withId(R.id.editText_pass))
    .perform(ViewActions.typeText("1a2S3d4f5@"));
  Espresso.onView(ViewMatchers.withId(R.id.button_login))
    .perform(ViewActions.click());
  Espresso.onView(ViewMatchers.withId(R.id.errorMessage))
    .check(ViewAssertions.matches(
      Matchers.not(ViewMatchers.isDisplayed())));
    }
```

Hands-On **(UI Testing)**: *Login Activity Instrumentation Test* - Test 3

```
@Test
public void errorMsgTextViewNotDisplayed(){
  Espresso.onView(ViewMatchers.withId(R.id.errorMessage))
    .check(ViewAssertions.matches(Matchers.not(
      ViewMatchers.isDisplayed())));
}
```

Mocking

Mockito - Definition

The core idea of unit tests is to test a unit of code in isolation from any of its dependencies. When you are unit testing a piece of code, you really don't care if its dependencies work correctly or not since you will need to test them separately anyway. Mockito or Easymock help you mock these dependencies.

testImplementation 'org.mockito:mockito-core:2.21.0'

Hands-On **(Mocking)**: *GeoCoder* - Unit Test

- @RunWith(MockitoJUnitRunner.class)
- We mock any class using @Mock, for example, @Mock private Geocoder geocoder;

Hands-On **(Mocking)**: *GeoCoder* - Set Up

```
@Before
   public void setUp(){
       geoUtils = new GeoUtils(geocoder);
   }
```

Hands-On **(Mocking)**: *GeoCoder* - Test (1)

```
public String getCountry(double lat, double lon)
throws IOException {
  List<Address> addresses =
    this.geocoder.getFromLocation(lat, lon, 1);
  return (addresses.size() > 0) ?
   addresses.get(0).getCountryName() : null;
    }
```

```
@Test
public void coordinatesWithNoCountryNameReturnNull()
throws Exception{
  String countryName = geoUtils.getCountry(0,0);
  assertNull(countryName);
}
```

Hands-On **(Mocking)**: *GeoCoder* - Test (2)

```
    @Test
    public void mordorLatLonReturnsMordor() throws Exception{

    //when we ask this address for the countryName return Mordor.
    when(addressForMordor.getCountryName()).thenReturn("Mordor");

    //when we ask for Mordor address we want this mock address
    when(geocoder.getFromLocation(
      anyDouble(), anyDouble(),anyInt()))
      .thenReturn(Arrays.asList(addressForMordor));

    String countryName = geoUtils.getCountry(0,0);
    assertEquals(countryName,"Mordor");
  }
```

That's all

QUESTIONS?

About me

www — jordimateofornes.com
github — github.com/JordiMateo
twitter — @MatForJordi
gdc — Distributed computation group