

DAM: Android Basic Testing

Applications for mobile devices & Course 2019-2020

Jordi Mateo Fornés jordi.mateo@udl.cat



- 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.com/JordiMateoUdL>



- **Applications for mobile devices.**
- Grau en Tècniques d'Interacció Digital i de Computació
- Campus Igualada - Escola Politècnica Superior - Universitat de Lleida
- All the code developed in this course can be found in this repository: DAM Course.

Agenda

- Introduction to testing
- Unit testing
 - Hands-On: Unit testing (Login Activity)
- Integration testing
 - Hands-On: Integration testing (Geocoder)
 - Hands-On: UI testing (Login Activity)
- Mocking:
 - Hands-On: Mocking Unit Test (Geocoder)

Introduction to testing

Why 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 right 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 little change on my code and get the feedback, why 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 need to run 10 times to ensure it work 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 cover 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

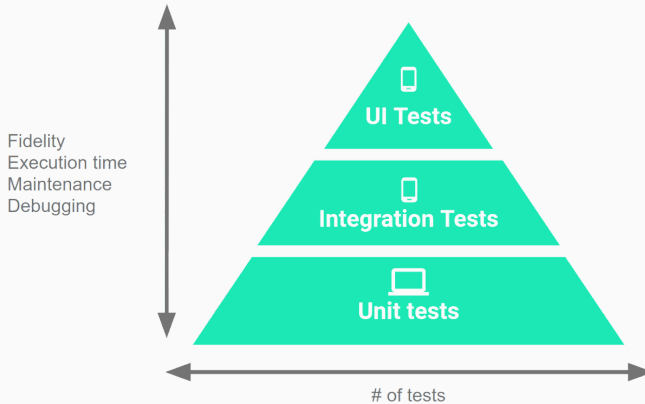


Figure 1: The Testing Pyramid, showing the three categories of tests that you should include in your app's test suite <https://developer.android.com/>

Unit Testing

What is Unit Testing

- Is a method to validate that each unit of the code 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. It provides assertions to identify test method. This tool test data first and then inserted in the piece of code.

Unit Testing: Pros and Cons

Advantages

- Easily verify that the logic of individual units is correct.
- Allows safe refactoring.
- We can use 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 a 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 **refactor**;
- @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, to avoid the try/catch we can use this throws Exception unless we want to test specific things with exceptions.

Hands-On (UnitTest): *LoginUtilsTest.class* - Do it yourself

- **TODO:** Add this function *isValidGmailAddress(String email)* and ensure it work properly adding unit tests.
- **TODO:** Refactor *LoginUtils* methods to be static, then update the tests.
- **TODO:** Add *getUserName(String email)* as the part before @, and ensure it work properly adding unit tests.

Integration Testing

What is an integration test (Instrumented)

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

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

Important tools

- Mockito.
- Espresso.
- Roboelectric (Next session).
- MockWebServer (Next session).

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!)

```
testInstrumentationRunner
"androidx.test.runner.AndroidJUnitRunner"
testImplementation 'junit:junit:4.12'
androidTestImplementation 'androidx.test.ext:junit:1.1.1'
androidTestImplementation
'androidx.test.espresso:espresso-core:3.2.0'
androidTestImplementation
'com.android.support.test:rules:1.0.2'
```

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

- Starts with the annotation **@RunWith(AndroidJUnit4.class)**, to indicated that is an instrumentation test and deploy the test 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);
    assertEquals("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);
}
```

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

androidTestImplementation "androidx.test.espresso:espresso-core:3.2.0"

`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.

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

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 properly 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 is all

Well done! Thanks for your attendance!

www — jordimateofoernes.com

github — github.com/JordiMateo

twitter — @MatForJordi

gdc — Distributed computation group