

Test Driven Development on Android Applications

mederic.hurier @ uni.lu



Agenda

- Motivation
- Concepts
- Unit Test
- UI Test
- Tutorial
- Demo



Motivation



To test or not to test, that is NOT the question ...

Ensuring that an application behaves as expected is **CRITICAL** to the people who are using it !

Risks associated to bugs found in software (ranked from least to most impactful):

- **School project:** bad grade / repeat year
- **Intranet portal:** disengagement from users
- **Web site / Mobile app:** 100 / 100K revenue
- **State / Banking system:** 100K / 100M revenue
- **Health care system (e.g. pacemaker):** Life or death
- **Military aircraft / Nuclear missile:** Massive destruction

Remember that the notion of “impact” is relative (people will remind you through your career)



The 11 most costly software errors in history

<https://raygun.com/blog/costly-software-errors-history/>

- EDS Child Support System (\$1 billion): highly complex and incompatible IT system
- Ariane 5 Flight 501 (\$500 million): integer overflow (64 bit number into 16 bit space)
- Bitcoin Hack on Mt. Gox (\$500 million, 5 years in prison): hack of a popular bitcoin exchange
- Pentium FDIV bug (\$475 million): flaw in a popular Intel Pentium processor, had to be replaced
- Knight's Error (\$440 million): flaw in trading algorithm that pushed erratic trades on 150 stocks
- NASA's Mars Climate Orbiter (\$125 million): failed conversion between English units to metric
- The Morris Worm (\$100 million cleanup, \$10,000 charge): "harmless experiment" from student
- The Mariner 1 Spacecraft (\$18 million): bug caused rocket to go back to earth (has self-destruct)
- Soviet Gas Pipeline Explosion: the CIA introduced a bug (largest non nuclear explosion on Earth)
- Patriot Missile Error (28 soldiers killed): inaccurate tracking calculation, become worse over time
- Heathrow Terminal 5 Opening (42,000 missed bag, 500 flights cancelled): software not scalable



Tools at your disposal to create better software

Question: How do you ensure that your application works as expected ?



Tools at your disposal to create better software

Question: How do you ensure that your application works as expected ?

- **print statements:** easy to add (and to forget) / must run manually
- **tracing / logging:** can be saved or shared / not easy to reproduce
- **type system:** strong guarantee at compile time / not complete
- **math proof:** strong guarantee overall / only for trivial program
- **debugging:** detailed program execution / must run manually
- **contracts:** strong guarantee at run time / not at compile time
- **tests:** can verify user's use cases / cannot test every use case

All these solutions have a value and a cost
A programmer must learn to balance their usage



In practice

During development:

- **Use logging and tracing** to monitor that goes in and out of method calls (print is quick and dirty)
- **Start the debugging system** if you need to gather better understanding on the program execution
- **Include assertion statements** to provide strong guarantees at run time, but disable on production
- **Create test methods** to verify that your use cases are satisfied (and to document program usage)

During maintenance:

- **You can leverage the techniques** you put in place during development to avoid future mistakes
 - e.g. are your tests still satisfied by your implementation ? if not, understand why with logs and assertions
- **Tests are a critical part of this workflow**, as they ensure that user's use cases remain valid



Concepts



Definitions

Unit Tests generally test the smallest possible unit of code without dependencies on system resources

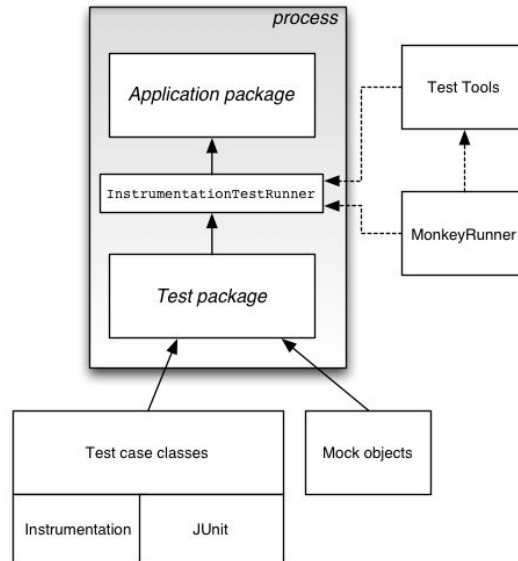
Test Cases define a set of objects and methods to run multiple tests independently from each other

Test Suites organize Test cases and can run them programmatically, in a repeatable manner

Test Runners orchestrate test execution and are provided by a **Testing Framework** (JUnit)

Test Fixtures consist of objects that must be initialized for running one or more tests

Dependency diagram





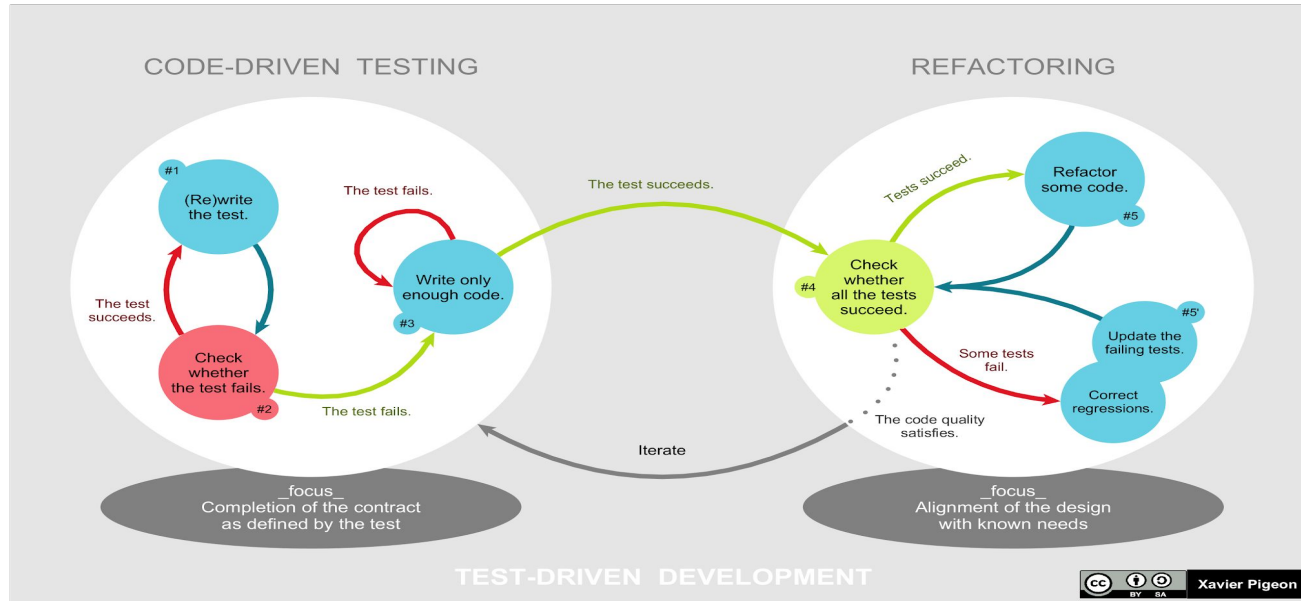
Components of a test case

In order to verify that there are no regressions in the functional behavior of your application,
it's important to create a test for each important Class of your project

For each test, you need to create the individual parts of a test case, including:

- test fixtures
- preconditions
- test methods
- postconditions

Development workflow





Best practices

Test Driven Development can help you catch bugs early (and later during maintenance)

Test Driven Development is encouraged by most development strategies in our industry:

- **Agile Software Development:** <http://agilemanifesto.org/>
 - SCRUM: <https://www.atlassian.com/agile/scrum>
 - XP Programming: <http://www.extremeprogramming.org/>
- **Test Apps on Android:** <https://developer.android.com/training/testing/>
 - Advanced Development: <https://eu.udacity.com/course/advanced-android-app-development--ud855>



Unit Test



Unit tests on Android

Unit Tests for Android applications are written in a structured way:

- Create a distinct class per test case and a test case for each class in your Android application
- Put your tests in a separate package, distinct from the code under test (e.g. `androidTest` or `test`)
- Package name should have the same name as the package you want to test, suffixed with `".tests"`
- Test case name should have the same name as the class you want to test, suffixed with `"Test"`



Where should run your tests ?

- **Local tests:** tests that run on your local machine to minimize execution time and resource usage
 - use this approach to run unit tests that have no dependencies on the Android framework
 - or have dependencies that can be filled by using mock objects (e.g. geolocation)
- **Instrumented tests:** tests that run on an Android device or emulator to access their resources
 - These tests have access to instrumentation information, such as the Context for the app under test.
 - use this approach to run tests that have Android dependencies which cannot be filled by using mock



What should you test ?

You should build tests when you need to verify the logic of some specific / independent code

For example:

if you are unit testing a class, your test might check that the class is in the right state after a method call

Unit tests are not suitable for testing complex scenario, such as interaction with the User Interface !

Instead, you should use the UI testing frameworks



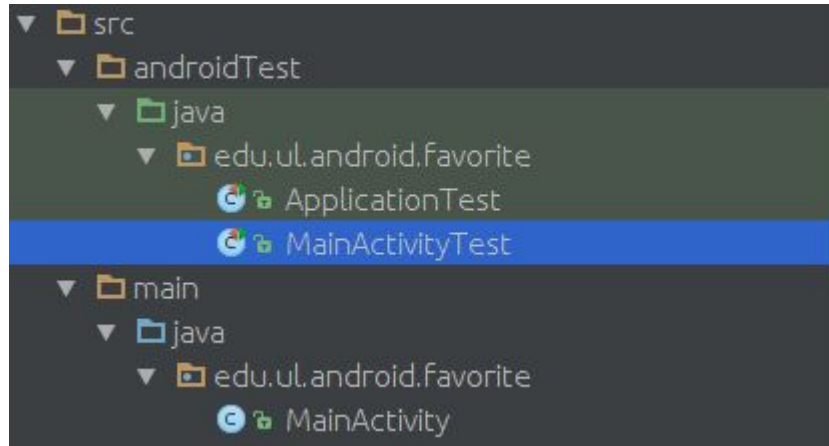
Testing frameworks for Android

- **Android Testing Library:** provides APIs that allow you to build and run test code for your apps
- **AndroidJUnitRunner:** a test runner compatible with JUnit, a popular framework on Java platform
- **Espresso:** an UI testing framework that uses a programmatic approach to test single application
- **UI Automator:** an UI testing framework suitable for cross-app testing across installed application
- **Monkey:** generates pseudo-random streams of user events such as clicks, touches, or gestures

Test creation on Android Studio

You need to create a Java Class inside the test package (created by default by Android Studio)

Here is an example with MainActivity:





Test definition with JUnit

Use the decorator `@Test` to annotate Test Methods:

```
import org.junit.*;

@RunWith(AndroidJUnit4.class)
public class TestFoobar {
    @Test
    public void testOneThing() {
        // Code that tests one thing
    }

    @Test
    public void testAnotherThing() {
        // Code that tests another thing
    }
}
```


`@Test(timeout=<milliseconds>)`: Specifies a timeout period for the test



Pre and Post execution

- **@Before:** will be invoked before each test
 - you can have multiple @Before methods
- **@After:** will be called after every test method
 - you can define multiple @After methods
- **@BeforeClass:** will be invoked once at initialization
 - your can have a single @BeforeClass methods
- **@AfterClass:** will be invoked once at finalization
 - your can have a single @AfterClass methods

Note: the order in which pre/post condition methods are executed is not fixed !



```
import org.junit.*;

@RunWith(AndroidJUnit4.class)
public class TestFoobar {
    @BeforeClass
    public static void setUpClass() {}

    @Before
    public void setUp() {}

    @After
    public void tearDown() {}

    @AfterClass
    public static void tearDownClass() {}
}
```



Precondition test

As a sanity check, it is good practice to verify that the test fixture has been set up correctly

example: the objects that you want to test have been correctly instantiated or initialized

By convention, the method for verifying your test fixture is called `testPreconditions()`

```
public void testPreconditions() {  
    assertNotNull("activity is null", activity);  
    assertNotNull("text is null", text);  
}
```




Test assertions

You can use Test Assertions to verify a specific condition:

- If the condition is true, the test passes and the assertion method does not do anything
- If the condition is false, the assertion method throws an AssertionError Exception
- In both cases, the test runner proceeds to run the other test methods in the test case

Note: you can provide a string in the first argument of your assertion to give some contextual details

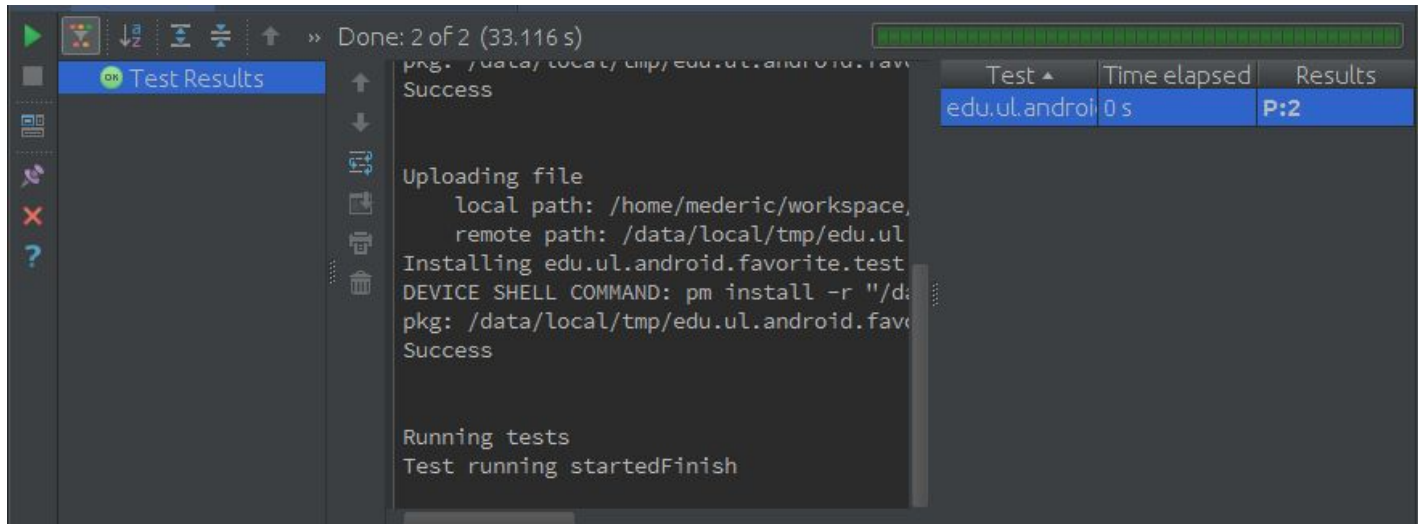


Assertion methods

- void **assertEquals**(boolean expected, boolean actual): check that two Objects are equal
- void **assertTrue**(boolean expected, boolean actual): check that a condition is true
- void **assertFalse**(boolean condition): check that a condition is false
- void **assertNotNull**(Object object): check that an object isn't null
- void **assertNull**(Object object): check that an object is null
- void **assertSame**(boolean condition): check if two references point to the same object
- void **assertNotSame**(boolean condition): check if two references not point to the same object
- void **assertArrayEquals**(expectedArray, resultArray): check whether two arrays are equal

Test execution

Right click on your test folder > Run tests in ...



The screenshot shows an IDE interface with a 'Test Results' panel on the left and a log window on the right. The 'Test Results' panel shows a green 'OK' icon and the text 'Test Results'. The log window displays the following text:

```
Done: 2 of 2 (33.116 s)
pkg: /data/local/tmp/edu.ul.android.favorite.test
Success

Uploading file
  local path: /home/mederic/workspace,
  remote path: /data/local/tmp/edu.ul
Installing edu.ul.android.favorite.test
DEVICE SHELL COMMAND: pm install -r "/d
pkg: /data/local/tmp/edu.ul.android.fave
Success

Running tests
Test running startedFinish
```

Test	Time elapsed	Results
edu.ul.android	0 s	P:2



UI Test



UI Testing on Android

User Interface (UI) testing lets you ensure that your app provides the interaction your expect

set a high standard of quality for your app such that it is more likely to be successfully adopted by users





Beta testing or Automated testing ?

One approach to UI testing is to simply have a human tester perform a set of user operations

However, this manual approach can be time-consuming, tedious, and error-prone

A more efficient approach is to write your UI tests that are performed in an automated way

The automated approach allows you to run your tests quickly and reliably in a repeatable manner.



White box or Black box ?

White-Box Testing: you have the source code for the application that you want to test

The Android Instrumentation framework is suitable for creating white-box tests for UI components

Black-Box Testing: you may not have access to the application source (only a compiled version)

This type of testing is useful when you want to test how your app interacts with external apps



UI testing on a single application

This type of test verifies the target app behaves as expected when a user performs a specific action

It allows you to check that the app returns the correct output in response to user interactions.

UI testing frameworks like Espresso allow you to programmatically simulate user actions:

<https://developer.android.com/training/testing/ui-testing/espresso-testing.html>



UI testing on a multiple applications

These tests verifies the correct behavior of interactions between different user apps or between apps

You might want to test that your camera app shares images correctly with a 3rd-party social media app

The **UI Automator framework** supports cross-app interactions and allow you to create such scenarios:

<https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html>



Example for an activity

To create a unit test for your Activity, use a `@Rule` decorator on its class:

```
@RunWith(AndroidJUnit4.class)
public class MainActivityTest {

    @Rule
    public ActivityTestRule<MainActivity> activity
        = new ActivityTestRule<>(MainActivity.class);

    @Test
    public void exampleTest() {
    }
}
```



More test annotations

These annotations let you classify how long a test should take to determine its execution frequency:

- **@SmallTests** (execution time: < 100ms): these tests should be run very frequently
- **@MediumTests** (execution time: < 2s): these tests run at least on every check in of your code
- **@LargeTests** (execution time: < 120s): these tests should be run as often as practical (once a day)



Example of a UI test

You can add a test method like this to test your UI:

```
@Test
public void topAndBottomWidgetsAreDisplayed() {
    onView(withId(R.id.ip_label))
        .check(matches(isDisplayed()));
    onView(withId(R.id.hex_check))
        .check(matches(isDisplayed()));
}
```



Tutorial



Goal

You have an existing Android App called "Favorite"

<https://github.com/fmind/favorite>

You have to register your favorite IP address, binary numbers, and hexadecimal numbers

Unfortunately, there is no tests for this app. You must create them to improve its quality



Scope

Included:

- Unit and UI/Activity testing
- Mock of Android framework
- Introduction to Monkey testing

Excluded:

- Service/Content provider testing
- Performance testing



Favorite

Favorite IP Address

CHECK IP ADDRESS

☐ IP Address valid ?

Favorite Binary Number

CHECK BINARY VALUE

☐ Binary Value valid ?

Favorite Hexadecimal Value

CHECK HEXADECIMAL VALUE

☐ Hexadecimal Value valid ?



Step 1: Instrumented Testing

1. Fill the methods in main **Validators.java**
2. Complete the units test in **ValidatorsTest.java**
3. Ask me to come and try to mess with your tests ;)

If your computer is not powerful enough to run an emulator

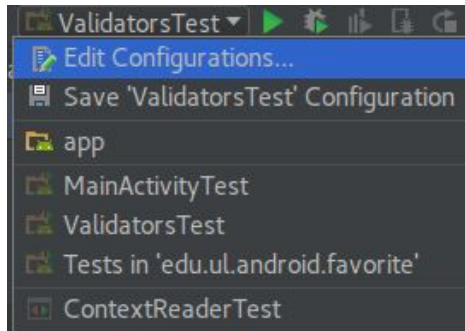
convert them to local tests and run them on your machine

move the class from `app/src/androidTest` to `app/src/test`

Run instrumented tests on Android Studio

1. Right-click on the file and select "Run ClassTest"
 - a. this works for the navigation menu and in the editor
 - b. you can also right click on a test method specifically

Then, you can run the same test from the top menu





Step 2: Activity Testing

1. Open **MainActivityTest.java** and fill the tests
2. Check that the text inputs are empty by default
3. Check that checkboxes are unchecked by default
4. Check that checkboxes are not clickable by default
5. Check the validation when a user submit an IP Address



Step 3: Local Unit Testing

1. Open ContextReaderTest.java and fill the test methods
2. Mock calls to Context methods from ContextReader

if it helps, look at the class imports and this link:

Document for Mokito on: <https://site.mockito.org>

Migrate your test class for Validators to become local test

since the test are local, they should be faster to execute



Step 4: Monkey Testing

Run the following code in your shell:

Multiple run will produce different output

Did you discover errors based on this framework ?

```
adb shell monkey -p edu.ul.android.favorite -v 1500
```



Demo