Mobile Testing
Today we are going to present to you about mobile testing.

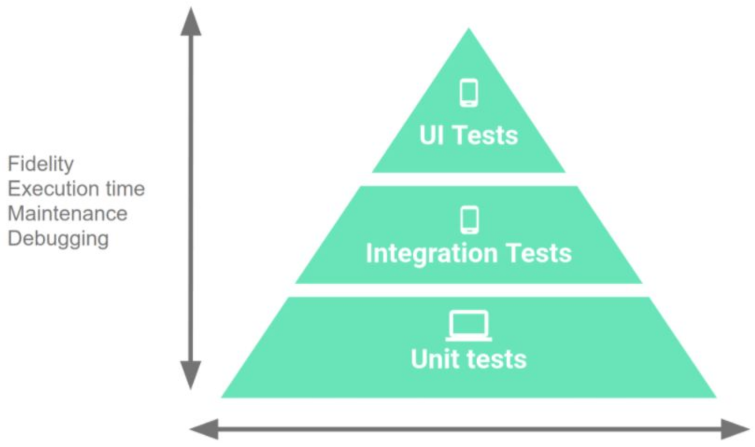
Fundamentals of testing
When creating a project, the most important thing is that the end product will work. The end user need to have a functioning system that works well and achieves the requirements. As a developer it is important to test all the functions that are wanted.

When a system is under development it is always expanding with new functions and therefore it is a good idea for the developers to continuously test the project. The regular way to tackle this is to use an iterative process which means that the developer and user have an ongoing discussion with each other during the process. This process makes it possible to change, delete and add things during the development and they can go back and change early steps even late in the process.

Testing also provides you with the following advantages:

- **Gives 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.

When testing your application, you can choose between:
- Testing on a **real device**
  - Highest fidelity/accuracy, but takes the most time to run.
- **Virtual device** (ex. Android Studio Emulator)
  - Improves the speeds of the testing at the expense of fidelity.
- **Simulated device** (ex. Robolectric)
  - Offers a balance between speed and fidelity when testing.



As can be seen in the picture, Unit tests is the biggest part of the testing process when developing an application. The pyramid is divided into three parts, small tests, medium tests and large tests. Small tests are included in the unit tests and should be run quickly with much focus. They are going to run locally on the users machine which makes it a local unit test which we will dig deeper into. It tests the functionality of small units of code in a repetitive way, for example methods, classes or components.

Google's internal test experts recommends the 70, 20, 10 rule, as the ratio between small, medium and large tests.

User Interface Tests (UI Tests)
Simulates a person's operations in the application. Pressing buttons and entering text.
Test if the user interaction is smooth and stable at 60 frames per second (fps), without dropped or delayed frames.
UI test need to be done with Android framework, therefore you have to run it on a real device or emulator.
The test is run with Espresso. Which is a Android test framework for user interface tests, intended to test single application.
You can find it in the directory "androidTest".

Integration Tests
Just as UI Tests, Integration tests run on an emulator or a real device. The reason to that is that the integration test is done to test the code with parts outside your own application. For example other apps or the software on a device. Some general facts about integration tests are:

- They run reasonably fast. Around a second up to a minute.
- Can access a local database, filesystem or network
- Shouldn't be used to practice Test Driven Development
- Can find problems that have not been found in the unit tests and give a more realistic test environment.

The last type are what we are going to focus on today, a type of test called Unit tests. **← Switch talker**

Unit tests
In order to build a successful program, unit tests are one of the most important parts. To verify if the code and the logic of units is right, you need to create and run unit tests against the code. After every build, it is important to run the tests to detect setbacks after the code has been changed.
Unit testing is also made to prevent the developer from writing bad and inaccurate code.

Unit tests should be built when the logic of some code in the app needs to be checked. For example, when testing a class, unit testing will see if the code is in the right state.
The code unit is usually tested in isolation, which means that the test affects and controls that unit only. To do this, providers such as Robolectric or a mocking framework like (Mockito) is used to seclude/isolate the unit from the dependencies. Or in easier terms, to only test a small part of the code.
Mock objects are objects that can mimic the behavior of real objects. Typically to test the behaviors of objects. For example, when car manufacturers simulates a crash, they use a crash dummy for the simulation, not a real person.
Unit tests are not the best for testing complex UI interaction events. For that, you should use the UI testing frameworks.

Two different types

Local unit tests
The local unit tests are a part of the small test as you saw from the pyramid. It runs locally on the users machine. As previously mentioned, they isolate code to test a small part of it. It runs on Java Virtual Machine to have a short runtime. If the tests needs objects in Android Framework, Robolectric is recommended. Tests that depends on the users dependencies, then mock objects are recommended, they emulate the dependencies behavior.

Create local unit test class
When using test classes, they should be written as a JUnit 4 class. JUnit is java-based and is the most used testing framework. It lets the user to write tests in a better way than the older JUnits. In order to create one of those, a class that contains one or more test methods need to be made.
Test name should also be descriptive.

Test methods begin with @Test annotation

Example of test method
```
import com.google.common.truth.Truth.assertThat
import org.junit.Test

class EmailValidatorTest {
    @Test
    fun emailValidator_CorrectEmailSimple_ReturnsTrue() {
        assertThat(EmailValidator.isValidEmail("name@email.com")).isTrue()
    }
}
```

About picture above
The "EmailValidatorTest" is the testclass for another class called "EmailValidator". The function under the @Test annotation, emailValidator_CorrectEmailSimple_ReturnsTrue controls if isValidEmail() below gets the correct result or not.

Include framework dependencies
ANDROIDX can provide Robolectric artefacts which should be used when the test interacts in a complex way or with many Android framework dependencies.

```
android {
    // ...
    testOptions {
        unitTests.includeAndroidResources = true
    }
}
```

Instrumented Unit tests
The instrumented unit tests are run on physical devices or emulators. They have access to the context for the app during test. An instrumented test can use the Android framework APIs, ANDROIDX for example. Compared to local tests, instrumented are much more faithful but much slower when running. Instrumented tests are therefore only recommended when it has to be tested against the behavior of a real device.

When needed, ANDROIDX can give the user libraries that simplifies the writing of instrumented unit tests. Android Builder is one of them, included in the library and helps the users to make it easier to create data objects.

In order to set up an instrumented test, you have to add ANDROIDX test APIs, they allow a faster solution for the user, to build and run the code for the apps. A JUnit 4 test runner is included in ANDROIDX, as well as APIs for functional user interface tests.

Create instrumented unit test class
Similar to the local unit test, the instrumented should also be a JUnit 4 class.

Create a test suite
In a test suite class, the user can group a number of test classes, to organize the

execution of the instrumented unit tests. They can also be nested, which means that the test suite can include others and run all the test classes together.

The test suite is included in the test package, almost like the main application package. It ends with .suite suffix (example, com.example.android.testing.project.suite). To create one for unit tests, the JUnit Runwith and suite classes have to be imported. Then @Runwith(suite.class) and (suite.suitClasses() annotations have to be added in the test suite. And the test classes or test suites have to be listed as arguments in the (Suite.SuiteClasses() annotation.

```
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)
class UnitTestSuite
```

The example here shows how a test suite called UnitTestSuite can be implemented. What that does is to run the classes CalculatorInstrumentationTest and CalculatorAddParameterizedTest together.

-------CODE EXAMPLE-------

Conclusion
In conclusion, mobile testing can be done in many different ways and it is a crucial part when it comes to software development. Without testing, the developers don’t know what works and what needs to be fixed, and the customers won’t be satisfied. As previously mentioned, the many different ways to test the application and code simplifies the process of developing software. Since it’s a complex business with many different components that have to cooperate.