Introduction to Android App Testing

Dr. Durga Prasad Mohapatra
Professor
Department of Computer Science & Engineering
N. I. T., Rourkela

Why, what, how, and when to test?

- Early bug detection saves a huge amount of project resources and reduces software maintenance costs. This is the best known reason to write tests for software development project.

- Writing tests will give you a deeper understanding of the requirements and the problem to be solved.

- The reason behind the approach of writing tests is to clearly understand the legacy or third-party code and having the testing infrastructure to confidently change or update the codebase.

- The more the code is covered by your tests, the higher the likelihood of discovering hidden bugs.

What to test

- Developer should test every statement in your code, but this also depends on different criteria and can be reduced to testing the main path of execution or just some key methods.
- Areas of android testing that should be considered are:
    - ➢Activity lifecycle events: You should test whether your activities handle life cycle events correctly. Configuration change events should also be tested as some of these events cause the current activity to be recreated.
    - ➢Database and file system operations: These operations should be tested to ensure that the operations and any errors are handled correctly. These operations should be tested in isolation at lower level, at a higher level or from the application itself.

- Physical characteristics of the device: Before shipping your application, you should be sure that all of the different devices it can be run on are supported, or at least you should detect the unsupported situations and take remedial actions. The characteristics of the devices that should be tested are:
  - Network capabilities
  - Screen densities
  - Screen resolutions
  - Screen sizes
  - Avalaibility of sensors
  - Keyboard & other input devices
  - GPS
  - External storage

- In this respect, an Android emulator can play an important role because it is practically impossible to have access to all of the devices with all of the possible combinations of features, but you can configure emulators for almost every situation.
- However, leave your final tests for actual devices where the real users will run the application so you get feedback from a real environment

Types of tests

- Testing comes in a variety of frameworks with differing levels of support from the Android SDK and your IDE of choice.
- There are several types of tests depending on the code being tested. Regardless of its type, a test should verify a condition and return the result of this evaluation as a single Boolean value that indicates its success or failure.
- Types of tests are:
  - ➢Unit tests
  - ➢Integration tests
  - ➢UI tests
  - ➢Functional or acceptance tests
  - ➢Performance tests
  - ➢System tests

Unit tests

- Unit tests are tests written by programmers for other programmers, and they should isolate the component under tests and be able to test it in a repeatable way.

- Thus, unit tests and mock objects are usually placed together. Mock object is a drop-in replacement for the real object, where you have more control of the object's behavior.

- Mock objects are used to isolate the unit from its dependencies, to monitor interactions, and also to be able to repeat the test any number of times.

- JUnit is the de facto standard for unit tests on Android. It's a simple open source framework for automating unit testing, originally written by Erich Gamma and Kent Beck.

Components used to build up a test case.

- The setUp() method: This method is called to initialize the fixture (fixture being the test and its surrounding code state). Overriding it, you have the opportunity to create objects and initialize fields that will be used by tests. It's worth noting that this setup occurs before every test.
- The tearDown() method: This method is called to finalize the fixture. Overriding it, you can release resources used by the initialization or tests. Again, this method is invoked after every test.
- For example, you can release a database or close a network connection here. There are more methods you can hook into before and after your test methods, but these are used rarely, and will be explained as we bump into them.

Components used to build up a test case.

- Outside the test method: JUnit is designed in a way that the entire tree of test instances is built in one pass, and then the tests are executed in a second pass.
  - ➢ This means that for very large and very long test runs with many Test instances, none of the tests may be garbage collected until the entire test is run.
  - ➢ This is particularly important in Android and while testing on limited devices as some tests may fail not because of an intrinsic failure but because of the amount of memory needed to run the application, in addition to its tests exceeding the device limits.

- Inside the test method: All public void methods whose names start with test will be considered as a test. As opposed to JUnit 4, JUnit 3 doesn’t use annotations to discover the tests; instead, it uses introspection to find their names.
  - ➢ There are some annotations available in the Android test framework such as @SmallTest, @MediumTest, or @LargeTest, which don’t turn a simple method into a test but organize them in different categories. Ultimately, you will have the ability to run tests for a single category using the test runner.

Components used to build up a test case.

- As a rule of thumb, name your tests in a descriptive way and use nouns and the condition being tested. Also, remember to test for exceptions and wrong values instead of just testing positive cases.
- For example, some valid tests and naming could be:
  - ➢testOnCreateValuesAreLoaded()
  - ➢testGivenIllegalArgumentThenAConversionErrorIsThrown()
  - ➢testConvertingInputToStringIsValid()

- During the execution of the test, some conditions, side effects, or method returns should be compared against the expectations. To ease these operations, JUnit provides a full set of assert* methods to compare the expected results from the test to the actual results after running them, throwing exceptions if the conditions are not met.
- Then, the test runner handles these exceptions and presents the results.

Components used to build up a test case.

- These methods, which are overloaded to support different arguments, include:
  - ➢assertTrue()
  - ➢assertFalse()
  - ➢assertEquals()
  - ➢assertNull()
  - ➢assertNotNull()
  - ➢assertSame()
  - ➢assertNotSame()
  - ➢fail()

Mock objects

- Mock objects are mimic objects used instead of calling the real domain objects to enable testing units in isolation.
- The Android testing framework supports mock objects that you will find very useful when writing tests. You need to provide some dependencies to be able to compile the tests.
- There are also external libraries that can be used when mocking. Several classes are provided by the Android testing framework in the android.test.mock package:
  - ➢MockApplication
  - ➢MockContentProvider
  - ➢MockContentResolver
  - ➢MockContext
  - ➢MockCursor
  - ➢MockDialogInterface
  - ➢MockPackageManager
  - ➢MockResources

Integration tests

- Integration tests are designed to test the way individual components work together. Modules that have been unit tested independently are now combined together to test the integration.
- Usually, Android Activities require some integration with the system infrastructure to be able to run. They need the Activity lifecycle provided by the Activity Manager, and access to resources, the file system, and databases.
- The same criteria apply to other Android components such as Services or Content Providers that need to interact with other parts of the system to achieve their duty.
- In all these cases, there are specialized test classes provided by the Android testing framework that facilitates the creation of tests for these components.

UI tests

- User Interface tests test the visual representation of your application, such as how a dialog looks or what UI changes are made when a dialog is dismissed.
- As you may have already known, only the main thread is allowed to alter the UI in Android. Thus, a special annotation @UIThreadTest is used to indicate that a particular test should be run on that thread and it would have the ability to alter the UI.
- On the other hand, if you only want to run parts of your test on the UI thread, you may use the *Activity.runOnUiThread(Runnable r)* method that provides the corresponding Runnable, which contains the testing instructions.

UI tests

- A helper class *TouchUtils* is also provided to aid in the UI test creation, allowing the generation of the following events to send to the Views, such as:
  - ➢Click
  - ➢Drag
  - ➢Long click
  - ➢Scroll
  - ➢Tap
  - ➢Touch

  By these means, you can actually remote control your application from the tests.

Functional or acceptance tests

- In agile software development, functional or acceptance tests are usually created by business and Quality Assurance (QA) people, and expressed in a business domain language.

- These are high-level tests to assert the completeness and correctness of a user story or feature.

- They are created ideally through collaboration between business customers, business analysts, QA, testers, and developers. However, the business customers (product owners) are the primary owners of these tests.

- Lately, within acceptance testing, a new trend named *Behavior-driven Development* has gained some popularity, and in a very brief description, it can be understood as a cousin of Test-driven Development.

Functional or acceptance tests

- Behavior-driven Development can be expressed as a framework of activities based on three principles (more information can be found at http://behaviour-driven.org):

  - ➢Business and technology should refer to the same system in the same way
  - ➢Any system should have an identified, verifiable value to the business
  - ➢Upfront analysis, design, and planning, all have a diminishing return

  To apply these principles, business people are usually involved in writing test case scenarios in a HLL and use a tool such as *jbehave*.

Test Case Scenario for “*jbehave*”

- Example:for this scenario is- Given I am using temperature converter. 100 Celsius should be converted to 212 Fahrenheit in respective field.
- *@Given(“I am using the temperature converter”)*

public void createTemperatureConverter(){

// do nothing this is syntactic sugar for readability }

@When(“enter Celsius”) {this.celsius=celsius}

@Then(“ObtainedFahrenhiet”) {assertEquals(Fahrenheit,TemperatureConverter.celsiusToFahrenheit(celsius)}

Performance tests

- Performance tests measure performance characteristics of the components in a repeatable way.
- If performance improvements are required by some part of the application, the best approach is to measure performance before and after a change is introduced.
- As is widely known, premature optimization does more harm than good, so it is better to clearly understand the impact of your changes on the overall performance.
- The introduction of the Dalvik JIT compiler in Android 2.2 changed some optimization patterns that were widely used in Android development.
- Nowadays, every recommendation about performance improvements in the Android developer’s site is backed up by performance tests.

System tests

- The system is tested as a whole, and the interaction between the components, software, and hardware is exercised. Normally, system tests include additional classes of tests such as:
    - ➢GUI tests
    - ➢Smoke tests
    - ➢Mutation tests
    - ➢Performance tests
    - ➢Installation tests

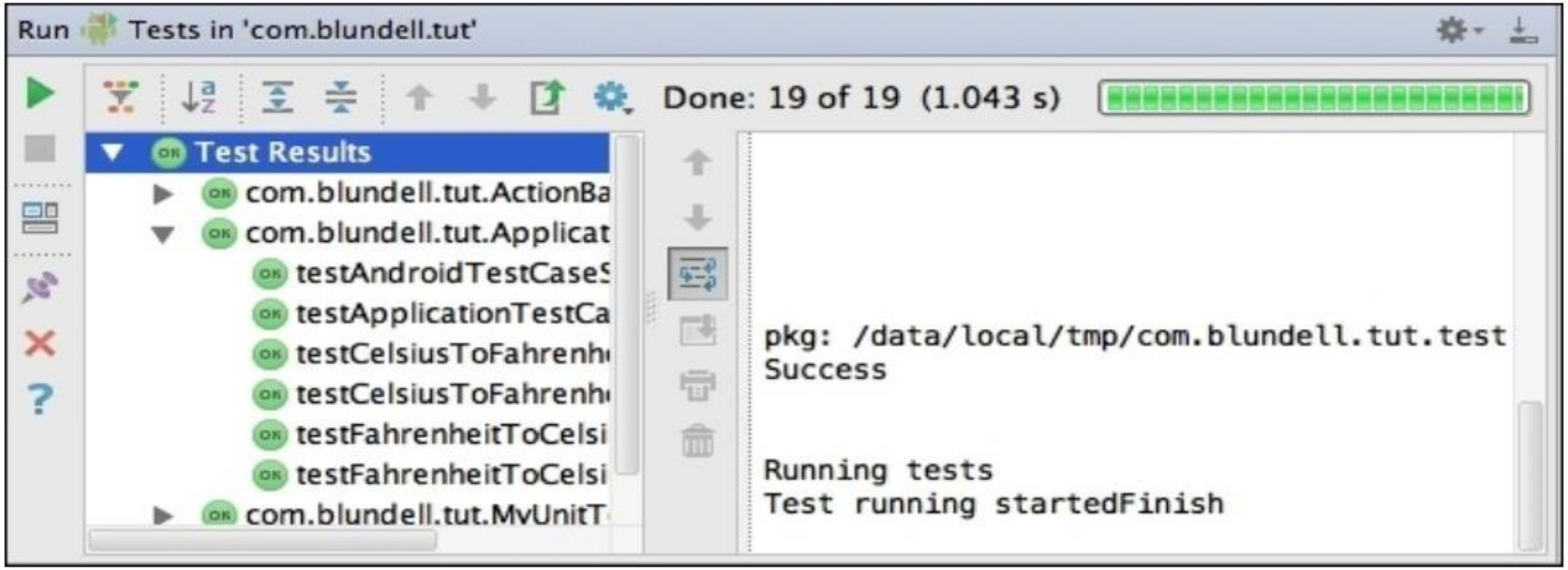
Smoke Testing

- To ensure that system testing would be meaningful.
- It is done before initiating system testing.
- If integrated program cannot pass even the basic tests, its not ready for vigorous testing
- A few test cases are designed to check basic functionalities working
- Ex- In library Automation system, smoke testing may check whether books can be created and deleted and so on.

Android Studio and IDE Support

- Junit is fully supported by this tool and helps in creating the tested android project.
- We can run the tests and analyze the results without leaving the IDE.
- We are able to run the tests from IDE allows us to debug tests that are not behaving correctly.
- Even if we are not developing in an IDE, we can find support to run the tests with gradle (check http://gradle.org if we are not familiar with this tool).

Continue…



Done: 19 of 19 (1.043 s)

- com.blundell.tut.ActionBa
- com.blundell.tut.Applicat
  - testAndroidTestCaseS
  - testApplicationTestCa
  - testCelsiusToFahrenh
  - testCelsiusToFahrenh
  - testFahrenheitToCelsi
  - testFahrenheitToCelsi
- com.blundell.tut.MyUnitT

```
pkg: /data/local/tmp/com.blundell.tut.test
Success


Running tests
Test running startedFinish
```

- Here we see Aside runs 19 unit tests, taking 1.043 seconds, with 0 errors and 0 failures detected.
- Name and duration of test is being displayed. Failure trace is shown on right side of screenshot showing test related information.

Java testing framework

- The Java testing framework is the backbone of Android testing, and sometimes, you can get away without writing Android-specific code.

- Android framework tests to a device, and this has an impact on the speed of our tests, that is, the speed we get feedback from a pass or a fail.

- If you architect your app in a clever way, you can create pure Java classes that can be tested in isolation away from Android.

- The two main benefits of this are increased speed of feedback from test results, and also, to quickly plug together libraries and code snippets to create powerful test suites, you can use the near ten years of experience of other programmers doing Java testing.

Android testing framework

- Android provides a very advanced testing framework that extends the industry standard JUnit library with specific features that are suitable to implement all of the testing strategies and types we mentioned before.
- In some cases, additional tools are needed, but the integration of these tools is, in most of the cases, simple and straightforward.
- Most relevant key features of the Android testing environment include:
  - ➢Android extensions to the JUnit framework that provide access to Android system objects
  - ➢An instrumentation framework that lets the tests control and examine the application
  - ➢Mock versions of commonly used Android system objects
  - ➢Tools to run single tests or test suites, with or without instrumentation
  - ➢Support to manage tests and test projects in Android Studio and at the command line

Instrumentation

- The instrumentation framework is the foundation of the testing framework.
- Instrumentation controls the application under tests and permits the injection of mock components required by the application to run. For example, you can create mock
- Contexts before the application starts and let the application use it.
- All the interactions of the application with the surrounding environment can be controlled using this approach.
- You can also isolate your application in a restricted environment to be able to predict the results that force the values returned by some methods, or that mock persistent and unchanged data for the ContentProvider's databases or even the file system content.

Instrumentation

- A standard Android project has its instrumentation tests in a correlated source folder called androidTest.

- This creates a separate application that runs tests on your application.

- There is no AndroidManifest here as it is automatically generated. The instrumentation can be customized inside the Android closure of your build.gradle file, and these changes are reflected in the autogenerated AndroidManifest.

- However, you can still run your tests with the default settings if you choose to change nothing.

Instrumentation

- Examples of things you can change are the test application package name, your test runner, or how to toggle performance-testing features:
  - ➢testApplicationId "com.blundell.something.non.default"
  - ➢testInstrumentationRunner "com.blundell.tut.CustomTestRunner"
  - ➢testHandleProfiling false
  - ➢testFunctionalTest true
  - ➢testCoverageEnabled true

- Here, the Instrumentation package (testApplicationId) is a different package to the main application. If you don’t change this yourself, it will default to your main application package with the .test suffix added.

Instrumentation

- At the moment, testHandleProfiling and testFunctionalTest are undocumented and unused, so watch out for when we are told what we can do with these.
- Setting testCoverageEnabled to true will allow you to gather code coverage reports using *Jacoco*.
- Also, notice that both the application being tested and the tests themselves are Android applications with their corresponding APKs installed.
- Internally, they will be sharing the same process and thus have access to the same set of features.

Gradle

- Gradle is an advanced build toolkit that allows you to manage dependencies and define a custom login to build your project.

- The Android build system is a plugin on top of Gradle, and this is what gives you the domain-specific language discussed previously such as setting a testInstrumentationRunner.

- The idea of using Gradle is that it allows you to build your Android apps from the command line for machines without using an IDE such as a continuous integration machine.

- Also, with first line integration of Gradle into the building of projects in Android Studio, you get the exact same custom build configuration from the IDE or command line.

Gradle

- Other benefits include being able to customize and extend the build process; for example, each time your CI builds your project, you could automatically upload a beta APK to the Google play store.

- You can create multiple APKs with different features using the same project, for example, one version that targets Google play in an app purchase and another that targets the Amazon app store’s coin payments.

Test targets

- During the evolution of your development project, your tests would be targeted to different devices.
- From simplicity, flexibility, and speed of testing on an emulator to the unavoidable final testing on the specific device you are intending your application to be run upon, you should be able to run your application on all of them.
- There are also some intermediate cases such as running your tests on a local JVM virtual machine, on the development computer, or on a Dalvik virtual machine or Activity, depending on the case.
- The emulator is probably the most powerful target as you can modify almost every parameter from its configuration to simulate different conditions for your tests.
- Ultimately, your application should be able to handle all of these situations, so it’s much better to discover the problems upfront than when the application has been delivered.

Creating the Android project

- We will create a new Android project. This is done from the ASide menu by going to File | New Project. This then leads us through the wysiwyg guide to create a project.
- In this particular case, we are using the following values for the required component names (clicking on the Next button in between screens):
  - ➢Application name: AndroidApplicationTestingGuide
  - ➢Company domain: blundell.com
  - ➢Form factor: Phone and Tablet
  - ➢Minimum SDK: 17
  - ➢Add an Activity: Blank Activity (go with default names)
- When you click on Finish and the application is created, it will automatically generate the androidTest source folder under the app/src directory, and this is where you can add your instrumented test cases.

Tip

- Alternatively, to create an androidTest folder for an existing Gradle Android project, you can select the src folder and then go to File | New | Directory.

- Then, write androidTest/java in the dialog prompt. When the project rebuilds, the path will then automatically be added so that you can create tests.

Package explorer

- After having created our project, the project view should look like one of the images shown in the following screenshot. This is because ASide has multiple ways to show the project outline.

- On the left, we can note the existence of the two source directories, one colored green for the test source and the other blue for the project source.

- On the right, we have the new Android project view that tries to simplify the hierarchy by compressing useless and merging functionally similar folders.

Creating a test cases

- As described before, we are creating our test cases in the src/androidTest/java folder of the project.
- You can create the file manually by right-clicking on the package and selecting New… |Java Class.
- However, in this particular case, we’ll take advantage of ASide to create our JUnit TestCase.
- Open the class under test (in this case, Main Activity) and hover over the class name until you see a light bulb (or press Ctrl/Command + 1). Select Create Test from the menu that appears.

Creating a test cases

- These are the values that we should enter when we create the test case:
    - ➢Testing library: JUnit 3
    - ➢Class name: MainActivityTest
    - ➢Superclass: junit.framework.TestCase
    - ➢Destination package: com.blundell.tut
    - ➢Superclass: junit.framework.TestCase
    - ➢Generate: Select none

- As you can see, you could also have checked one of the methods of the class to generate an empty test method stub. These stub methods may be useful in some cases, but you have to consider that testing should be a behavior-driven process rather than a method-driven one.

Creating a test cases

- The basic infrastructure for our tests is in place; what is left is to add a dummy test to verify that everything is working as expected. We now have a test case template, so the next step is to start completing it to suit our needs.
- To do it, open the recently created test class and add the testSomething() test.
- We should have something like this:

```
package com.blundell.tut;
import android.test.suitebuilder.annotation.SmallTest;
import junit.framework.TestCase;
public class MainActivityTest extends TestCase {
public MainActivityTest() {
super("MainActivityTest");
}
@SmallTest
public void testSomething() throws Exception {
fail("Not implemented yet");
}
}
```

Creating a test cases

- The no-argument constructor is needed to run a specific test from the command line, as explained later using an instrumentation.

- This test will always fail, presenting the message: Not implemented yet. In order to do this, we will use the fail method from the junit.framework.Assert class that fails the test with the given message.

Test annotations

- Looking carefully at the test definition, you might notice that we decorated the test using the @SmallTest annotation, which is a way to organize or categorize our tests and run them separately.

- This test will always fail, presenting the message: Not implemented yet. In order to do this, we will use the fail method from the junit.framework.Assert class that fails the test with the given message.

Test annotations

- There are other annotations that can be used by the tests, such as:

Annotation	**Description**
@SmallTest	Marks a test that should run as part of the small tests.
@MediumTest	Marks a test that should run as part of the medium tests.
@LargeTest	Marks a test that should run as part of the large tests.
@Smoke	Marks a test that should run as part of the smoke tests. The android.test.suitebuilder.SmokeTestSuiteBuilder will run all tests with this annotation.
@FlakyTest	Use this annotation on the InstrumentationTestCase class' test methods. When this is present, the test method is re-executed if the test fails. The total number of executions is specified by the tolerance, and defaults to 1. This is useful for tests that may fail due to an external condition that could vary with time. For example, to specify a tolerance of 4, you would annotate your test with: @FlakyTest(tolerance=4).

Test annotations

- There are other annotations that can be used by the tests, such as:

Annotation	**Description**
@UIThreadTest	Use this annotation on the InstrumentationTestCase class' test methods. When this is present, the test method is executed on the application's main thread (or UI thread). As instrumentation methods may not be used when this annotation is present, there are other techniques if, for example, you need to modify the UI and get access to the instrumentation within the same test. In such cases, you can resort to the Activity.runOnUIThread() method that allows you to create any Runnable and run it in the UI thread from within your test: mActivity.runOnUIThread(new Runnable() { public void run() { // do somethings } });
@Suppress	Use this annotation on test classes or test methods that should not be included in a test suite. This annotation can be used at the class level, where none of the methods in that class are included in the test suite, or at the method level, to exclude just a single method or a set of methods.

Running the tests

- There are several ways of running our tests, and we will analyze them here.
- Additionally, tests can be grouped or categorized and run together, depending on the situation.
- <u>Running all tests from Android Studio:</u>
  - ➢This is perhaps the simplest method if you have adopted ASide as your development environment. This will run all the tests in the package.
  - ➢Select the app module in your project and then go to Run | (android icon) All Tests. If a suitable device or emulator is not found, you will be asked to start or connect one.
  - ➢ The tests are then run, and the results are presented inside the Run perspective, as shown in the following screenshot:

Running the tests

- A more detailed view of the results and the messages produced during their execution can also be obtained in the LogCat view within the Android DDMS perspective, as shown in the following screenshot:
- Running a single test case from your IDE:

➢There is an option to run a single test case from ASide, should you need to. Open the file where the test resides, right-click on the method name you want to run, and just like you run all the tests, select Run | (android icon) testMethodName.

➢When you run this, as usual, only this test will be executed. In our case, we have only one test, so the result will be similar to the screenshot presented earlier.

Running the tests

- Running from the emulator:

- ➢The default system image used by the emulator has the Dev Tools application installed, providing several handy tools and settings. Among these tools, we can find a rather long list, as is shown in the following screenshot.
- ➢Now, we are interested in Instrumentation, which is the way to run our tests. This application lists all of the packages installed that define instrumentation tag tests in their project. We can run the tests by selecting our tests based on the package name, as shown in the following screenshot.

Running the tests

- Running tests from the command line:

➢ Finally, tests can be run from the command line too. This is useful if you want to automate or script the process.

➢ To run the tests, we use the am instrument command (strictly speaking, the am command and instrument subcommand), which allows us to run instrumentations specifying the package name and some other options.

➢ You might wonder what “am” stands for. It is short for Activity Manager, a main component of the internal Android infrastructure that is started by the System Server at the beginning of the boot process, and it is responsible for managing Activities and their life cycle. Additionally, as we can see here, it is also responsible for Activity instrumentation.

Running the tests

➢The general usage of the am instrument command is:

am instrument [flags] <COMPONENTS> -r -e <NAME> <VALUE> -p <FILE> -w

This table summarizes the most common options:

Option	**Description**
-r	Prints raw results. This is useful to collect raw performance data.
-e <NAME> <VALUE>	Sets arguments by name. We will examine its usage shortly. This is a generic option argument that allows us to set the <name, value> pairs.
-p <FILE>	Writes profiling data to an external file.
-w	Waits for instrumentation to finish before exiting. This is normally used in commands. Although not mandatory, it's very handy, as otherwise, you will not be able to see the test's results.

Running the tests

- Running all tests: This command line will open the adb shell and then run all tests with the exception of performance tests:

```
$: adb shell
#: am instrument -w
com.blundell.tut.test/android.test.InstrumentationTestRunner
com.blundell.tut.MainActivityTest: Failure in testSomething:
junit.framework.AssertionFailedError: Not implemented yet at
com.blundell.tut.MainActivityTest.testSomething(MainActivityTest.java:15) at
java.lang.reflect.Method.invokeNative(Native Method) at
android.test.AndroidTestRunner.runTest(AndroidTestRunner.java:191) at
android.test.AndroidTestRunner.runTest(AndroidTestRunner.java:176) at
android.test.InstrumentationTestRunner.onStart
(InstrumentationTestRunner.java:554) at
android.app.Instrumentation$InstrumentationThread.run
(Instrumentation.java:1701) Test results for InstrumentationTestRunner=.F Time:
0.002 FAILURES!!! Tests run: 1, Failures: 1, Errors: 0
```

Note that the package you declare with –w is the package of your instrumentation tests, not the package of the application under test.

Running the tests

- <u>Running tests from a specific test:</u>

➢To run all the tests in a specific test case, you can use:

```
$: adb shell
#:am instrument -w -e class com.blundell.tut.MainActivityTest com.blundell.tut.test/android.test.InstrumentationTestRunner
```

- <u>Running a specific test by name :</u>

➢Additionally, we have the alternative of specifying which test we want to run in the command line:

```
$: adb shell
#: am instrument -w -e class com.blundell.tut.MainActivityTest\#testSomething com.blundell.tut.test/android.test.InstrumentationTestRunner
```

➢ This test cannot be run in this way unless we have a no-argument constructor in our test case; that is the reason we added it before.

Running the tests

- Running specific tests by category:

➢As mentioned before, tests can be grouped into different categories using annotations (Test Annotations), and you can run all tests in this category.

➢ The following options can be added to the command line:

Option	**Description**
-e unit true	This runs all unit tests. These are tests that are not derived from InstrumentationTestCase (and are not performance tests).
-e func true	This runs all functional tests. These are tests that are derived from InstrumentationTestCase.
-e perf true	This includes performance tests.
-e size {small | medium | large}	This runs small, medium, or large tests depending on the annotations added to the tests.
-e annotation <annotation-name>	This runs tests annotated with this annotation. This option is mutually exclusive with the size option.

Running the tests

- In our example, we annotated the test method testSomething() with @SmallTest. So this test is considered to be in that category, and is thus run eventually with other tests that belong to that same category, when we specify the test size as small.

- This command line will run all the tests annotated with @SmallTest:

```
$: adb shell
#: am instrument -w -e size small
com.blundell.tut.test/android.test.InstrumentationTestRunner
```

Running the tests

- Running tests using Gradle:

➢ Your gradle build script can also help you run the tests and this will actually do the previous commands under the hood. Gradle can run your tests with this command:

```
gradle  connectedAndroidTest
```

Creating a custom annotation

In case you decide to sort the tests by a criterion other than their size, a custom annotation can be created and then specified in the command line.

As an example, let’s say we want to arrange our tests according to their importance, so we create an annotation @VeryImportantTest, which we will use in any class where we write tests (MainActivityTest for example):

Running the tests

```
package com.blundell.tut;
/**
* Marker interface to segregate important tests
*/
@Retention(RetentionPolicy.RUNTIME)
public @interface VeryImportantTest {
}
```

Following this, we can create another test and annotate it with @VeryImportantTest:

```
@VeryImportantTest
public void testOtherStuff() {
fail("Also not implemented yet");
}
```

So, as we mentioned before, we can include this annotation in the am instrument command line to run only the annotated tests:

```
$: adb shell
#: am instrument -w -e annotation com.blundell.tut.VeryImportantTest
com.blundell.tut.test/android.test. InstrumentationTestRunner
```

Running the tests

- <u>Running performance tests:</u>

We will be reviewing performance test details in Chapter 8, Testing and Profiling Performance, but here, we will introduce the available options to the am instrument command.

To include performance tests on your test run, you should add this command line option:

-e perf true: This includes performance tests

- <u>Dry run</u>

Sometimes, you might only need to know what tests will be run instead of actually running them.

This is the option you need to add to your command line:

-e log true: This displays the tests to be run instead of running them

This is useful if you are writing scripts around your tests or perhaps building other tools.

Debugging tests

- You should assume that your tests might have bugs too. In such a case, usual debugging techniques apply, for example, adding messages through LogCat.
- If a more sophisticated debugging technique is needed, you should attach the debugger to the test runner.
- In order to do this without giving up on the convenience of the IDE and not having to remember hard-to-memorize command-line options, you can Debug Run your run configurations. Thus, you can set a breakpoint in your tests and use it. To toggle a breakpoint, you can select the desired line in the editor and left-click on the margin.
- Once it is done, you will be in a standard debugging session, and the debug window should be available to you.
- It is also possible to debug your tests from the command line; you can use code instructions to wait for your debugger to attach.

Other command line options

- The am instrument command accepts other <name, value> pairs beside the previously mentioned ones:

Name	**Value**
debug	true. Set break points in your code.
package	This is a fully qualified package name of one or several packages in the test application.
class	A fully qualified test case class to be executed by the test runner. Optionally, this could include the test method name separated from the class name by a hash (#).
coverage	true. Runs the EMMA code coverage and writes the output to a file that can also be specified. We will dig into the details about supporting EMMA code coverage for our tests in Chapter 9, Alternative Testing Tactics.