Learning Android Application Testing

This book is a practical introduction to readily available techniques, frameworks, and tools to thoroughly test your Android applications and improve project development. You will learn the Java testing framework, how to create a test case and debug it. Next, you'll be walked through using the Android SDK to test using the ActivityTestCase and ActivityUnitTest classes as well as discussing popular testing libraries. Through examples you will test files, databases, ContentProviders, exceptions, services, and test your app using Espresso. You will discover how to manage your Android testing environment using Android emulators, deep dive into how adb and the emulator can super charge your testing automation, and also test user interactions with monkeyrunner. You will also be guided through different testing methodologies including Test-driven Development and Behavior-driven Development and will learn how to perform Unit and Functional testing applying them to your Android projects. You will also use continuous integration techniques for ultimate application quality control using Gradle and Jenkins.

By the end of the book, you'll be looking through alternative testing tactics including Fest and Spoon to build upon and expand your Android testing range and finesses.

Who this book is written for
If you are an Android developer looking to test your applications or optimize your application development process, then this book is for you. No previous experience in application testing is required.

What you will learn from this book
- Apply testing techniques and utilize tools to improve Android application development
- Get to grips with the nuances of testing on Android, including how to architect an application to facilitate better testing
- Explore the Android instrumentation testing framework to optimize your activities, services, content providers, and usage of other Android components
- Understand different development methodologies including Test-driven Development and Behavior-driven Development
- Apply the continuous integration technique for ultimate application quality control
- Improve application performance by analyzing the results returned from performance tests
- Expose your application to a wide range of conditions and configurations to simulate real-life network conditions and detect problems in the application
- Explore further tools to improve application quality such as micro benchmarks and code coverage


Paul Blundell
Diego Torres Milano

PUBLISHING

Community Experience Distilled

Learning Android Application Testing

Improve your Android applications through intensive testing and debugging
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'Getting Started with Testing'
- A synopsis of the book’s content
- More information on Learning Android Application Testing

About the Authors

Paul Blundell is an aspiring software craftsman and senior Android developer at Novoda. Before Novoda, he worked at AutoTrader and Thales, with apps that he released racking up over one million downloads. A strong believer in software craftsmanship, SOLID architecture, clean code, and testing, Paul has used this methodology to successfully nurture and create many Android applications. These include the Tesco launcher app, which was preinstalled for the recently released Hudl2 tablet; MUBI, a unique film streaming service; and the AutoTrader UK car search app.

If anyone wants to provide feedback, you can always tweet to him @blundell_apps. He also likes to write, so you can find more material at http://blog.blundellapps.com/.

Diego Torres Milano has been involved with the Android platform since its inception, by the end of 2007, when he started exploring and researching the platform's possibilities, mainly in the areas of user interfaces, unit and acceptance tests, and Test-driven Development.

This is reflected by a number of articles mainly published on his personal blog (http://dtmilano.blogspot.com), and his participation as a lecturer in some conferences and courses, such as Mobile Dev Camp 2008 in Amsterdam (Netherlands) and Japan Linux Symposium 2009 (Tokyo), Droidcon London 2009, and Skillsmatter 2009 (London, UK). He has also authored Android training courses delivered to various companies in Europe.
Previously, he was the founder and developer of several open source projects, mainly CULT Universal Linux Thin Project (http://cult-thinclient.sf.net) and the very successful PXES Universal Linux Thin Client project (that was later acquired by 2X Software, http://www.2x.com). PXES is a Linux-based operating system specialized for thin clients, used by hundreds of thousands of thin clients all over the world. This project has a popularity peak of 35 million hits and 400K downloads from SourceForge in 2005. This project had a dual impact. Big companies in Europe decided to use it because of improved security and efficiency; and organizations, institutions, and schools in some development countries in South America, Africa, and Asia decided to use it because of the minimal hardware requirements, having a huge social impact of providing computers, sometimes recycled ones, to everyone.

Among the other open source projects that he founded are Autoglade, Gnome-tla, and JGlade, and he has contributed to various Linux distributions, such as RedHat, Fedora, and Ubuntu.

He has also given presentations at the LinuxWorld, LinuxTag, GUADEC ES, University of Buenos Aires, and so on.

Diego has also developed software, participated in open source projects, and advised companies worldwide for more than 15 years.

He can be contacted at dtmilano@gmail.com.
Learning Android Application Testing

It doesn't matter how much time you invest in Android design, or even how careful you are when programming, mistakes are inevitable and bugs will appear. This book will help you minimize the impact of these errors in your Android project and increase your development productivity. It will show you the problems that are easily avoided, to help get you quickly to the testing stage.

Android Application Testing Guide is the first and only book providing a practical introduction to the most commonly available techniques, frameworks, and tools to improve the development of your Android applications. Clear, step-by-step instructions show how to write tests for your applications and assure quality control using various methodologies.

The author's experience in applying application testing techniques to real-world projects enables him to share insights on creating professional Android applications.

The book covers the basics of framework support for tests to architectures and techniques such as Test-driven Development, which is an agile component of the software development process and a technique where you will tackle bugs early on. From the most basic unit tests applied to a sample project to more sophisticated performance tests, this book provides a detailed description of the most widely used techniques in the Android testing world in a recipe-based approach.

The author has extensive experience of working on various development projects throughout his professional career. All this research and knowledge has helped create a book that will serve as a useful resource to any developer navigating the world of Android testing.

What This Book Covers

Chapter 1, Getting Started with Testing, introduces the different types of testing and their applicability to software development projects in general and to Android in particular. It then goes on to cover testing on the Android platform, unit testing and JUnit, creating an Android test project and running tests.

Chapter 2, Understanding Testing with the Android SDK, starts digging a bit deeper to recognize the building blocks available to create the tests. It covers Assertions, TouchUtils, which are intended to test user interfaces, mock objects, instrumentation, and TestCase class hierarchies.
Chapter 3, Baking with Testing Recipes, provides practical examples of different situations you will commonly encounter while applying the disciplines and techniques described before. The examples are presented in a cookbook style so you can adapt and use them for your projects. The recipes cover Android unit tests, activities, applications, databases, and ContentProviders, services, UIs, exceptions, parsers, memory leaks, and a look at testing with Espresso.

Chapter 4, Managing Your Android Testing Environment, provides different conditions to run the tests. It starts with the creation of the Android Virtual Devices (AVD) to provide different conditions and configurations for the application under test and runs the tests using the available options. Finally, it introduces monkey as a way to generate simulated events used for testing.

Chapter 5, Discovering Continuous Integration, introduces this agile technique for software engineering and automation that aims to improve the software quality and reduce the time taken to integrate changes by continuously applying integration and testing frequently.

Chapter 6, Practicing Test-driven Development, introduces the Test-driven Development discipline. It starts with a general revision and later on moves to the concepts and techniques closely related to the Android platform. This is a code-intensive chapter.

Chapter 7, Behavior-driven Development, introduces Behavior-driven Development and some concepts, such as the use of a common vocabulary to express the tests and the inclusion of business participants in the software development project.

Chapter 8, Testing and Profiling Performance, introduces a series of concepts related to benchmarking and profiles from traditional logging statement methods to creating Android performance tests and using profiling tools.

Chapter 9, Alternative Testing Tactics, covers adding code coverage to ensure you know what is tested and what isn't, as well as testing on the host's Java Virtual Machine, investigating Fest, Spoon, and the future of Android testing to build upon and expand your Android testing range.
Getting Started with Testing

Firstly, I will avoid introductions to Android since it is covered in many books already, and I am inclined to believe that if you are reading a book that covers this more advanced topic, you will have already started with Android development.

I will be reviewing the main concepts behind testing, and the techniques, frameworks, and tools available to deploy your testing strategy on Android.

After this overview, we can put the concepts learned into practice. In this chapter we will cover:

- Setting up the infrastructure to test on Android
- Running unit tests using JUnit
- Creating an Android instrumentation test project
- Running multiple tests

We will be creating a simple Android project and its companion tests. The main project will be bare bones so that you can concentrate on the testing components.

I would suggest that new developers with no Android testing experience read this book. If you have more experience with Android projects and have been using testing techniques for them, you might read this chapter as a revision or reaffirmation of the concepts.

**Why, what, how, and when to test?**

You should understand that 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 your software development project. Increased productivity will soon be evident.
Additionally, writing tests will give you a deeper understanding of the requirements and the problem to be solved. You will not be able to write tests for a piece of software you don’t understand.

This is also the reason behind the approach of writing tests to clearly understand 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.

If, during this coverage analysis, you find that some areas of your code are not exercised, additional tests should be added to cover this code as well.

To help in this request, enter Jacoco (http://www.eclemma.org/jacoco/), an open source toolkit that measures and reports Java code coverage. It supports various coverage types, as follows:

- Class
- Method
- Block
- Line

Coverage reports can also be obtained in different output formats. Jacoco is supported to some degree by the Android framework, and it is possible to build a Jacoco instrumented version of an Android app.

We will be analyzing the use of Jacoco on Android to guide us to full test coverage of our code in Chapter 9, Alternative Testing Tactics.

This screenshot shows how a Jacoco code coverage report is displayed as an HTML file that shows green lines when the code has been tested:
By default, the Jacoco gradle plugin isn't supported in Android Studio; therefore, you cannot see code coverage in your IDE, and so code coverage has to be viewed as separate HTML reports. There are other options available with other plugins such as Atlassian's Clover or Eclipse with EcLEmma.

Tests should be automated, and you should run some or all tests every time you introduce a change or addition to your code in order to ensure that all the conditions that were met before are still met, and that the new code satisfies the tests as expected.

This leads us to the introduction of Continuous Integration, which will be discussed in detail in Chapter 5, Discovering Continuous Integration, enabling the automation of tests and the building process.

If you don't use automated testing, it is practically impossible to adopt Continuous Integration as part of the development process, and it is very difficult to ensure that changes would not break existing code.

Having tests stops you from introducing new bugs into already completed features when you touch the code base. These regressions are easily done, and tests are a barrier to this happening. Further, you can now catch and find problems at compile time, that is, when you are developing, rather than receiving them as feedback when your users start complaining.
What to test

Strictly speaking, you 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. Usually, there's no need to test something that can't be broken; for example, it usually makes no sense to test getters and setters as you probably won't be testing the Java compiler on your own code, and the compiler would have already performed its tests.

In addition to your domain-specific functional areas that you should test, there are some other areas of an Android application that you should consider. We will be looking at these in the following sections.

Activity lifecycle events

You should test whether your activities handle lifecycle events correctly.

If your activity should save its state during the `onPause()` or `onDestroy()` events and later be able to restore it in `onCreate(Bundle savedInstanceState)`, then you should be able to reproduce and test all these conditions and verify that the state was correctly saved and restored.

Configuration change events should also be tested as some of these events cause the current Activity to be recreated. You should test whether the handling of the event is correct and that the newly created Activity preserves the previous state. Configuration changes are triggered even by a device rotation, so you should test your application's ability to handle these situations.

Database and filesystem operations

Database and filesystem operations should be tested to ensure that the operations and any errors are handled correctly. These operations should be tested in isolation at the lower system level, at a higher level through `ContentProviders`, or from the application itself.

To test these components in isolation, Android provides some mock objects in the `android.test.mock` package. A simple way to think of a mock is as a drop-in replacement for the real object, where you have more control of the object's behavior.

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 situation and take pertinent measures.
The characteristics of the devices that you should test are:

- Network capabilities
- Screen densities
- Screen resolutions
- Screen sizes
- Availability of sensors
- Keyboard and 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, as mentioned before, 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. For now, we are going to concentrate on how to test Android apps using the instrumented Android testing framework, which has full SDK and ASide support, and later on, we will discuss the alternatives.

Testing can be implemented at any time in the development process, depending on the test method employed. However, we will be promoting testing at an early stage of the development cycle, even before the full set of requirements has been defined and the coding process has been started.

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.
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. That's why unit tests and mock objects are usually placed together. You use mock objects to isolate the unit from its dependencies, to monitor interactions, and also to be able to repeat the test any number of times. For example, if your test deletes some data from a database, you probably don't want the data to be actually deleted and, therefore, not found the next time the test is ran.

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.

Android test cases use JUnit 3 (this is about to change to JUnit 4 in an impending Google release, but as of the time of this writing, we are showing examples with JUnit 3). This version doesn't have annotations, and uses introspection to detect the tests.

A typical Android-instrumented JUnit test would be something like this:

```java
public class MyUnitTestCase extends TestCase {

    public MyUnitTestCase() {
        super("testSomething");
    }

    public void testSomething() {
        fail("Test not implemented yet");
    }
}
```

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

The following sections explain the components that can be used to build up a test case. Note that these components and the pattern of working with a test case are not unique to unit tests, and they can be deployed for the other test types that we will discuss in the following sections.
**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.

**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. Therefore, the test runner holds strong references to all test instances for the duration of the test execution. 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.

Therefore, if you allocate external or limited resources in a test, such as Services or ContentProviders, you are responsible for freeing those resources. Explicitly setting an object to null in the tearDown() method, for example, allows it to be garbage collected before the end of the entire test run.

**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.
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.

These methods, which are overloaded to support different arguments, include:

- `assertTrue()`
- `assertFalse()`
- `assertEquals()`
- `assertNull()`
- `assertNotNull()`
- `assertSame()`
- `assertNotSame()`
- `fail()`

In addition to these JUnit assert methods, Android extends Assert in two specialized classes, providing additional tests:

- `MoreAsserts`
- `ViewAsserts`

**Mock objects**

Mock objects are mimic objects used instead of calling the real domain objects to enable testing units in isolation.

Generally, this is accomplished to verify that the correct methods are called, but they can also be of great help to isolate your tests from the surrounding code and be able to run the tests independently and ensure repeatability.
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

Almost any component of the platform that could interact with your Activity can be created by instantiating one of these classes.

However, they are not real implementations but stubs, the idea being you extend one of these classes to create a real mock object and override the methods you want to implement. Any methods you do not override will throw an `UnsupportedOperationException`.

**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 `ActivityManager`, and access to resources, the filesystem, and databases.

The same criteria apply to other Android components such as `Services` or `ContentProviders` 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.

Special considerations should be taken if your tests involve UI components. 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)` method that provides the corresponding `Runnable`, which contains the testing instructions.

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. Also, Android has recently introduced Espresso for UI instrumented tests, and we will be covering this in Chapter 3, Baking with Testing Recipes.

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.

Some frameworks and tools can help in this field, such as Calabash (http://calabash.sh) or most notably FitNesse (http://www.fitnesse.org), which can be easily integrated, up to some point, into the Android development process, and will let you create acceptance tests and check their results as follows:
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. It aims to provide a common vocabulary between business and technology people in order to increase mutual understanding.

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](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 high-level language and use a tool such as **jbehave** ([http://jbehave.org](http://jbehave.org)). In the following example, these scenarios are translated into Java code that expresses the same test scenario.
Test case scenario
As an illustration of this technique, here is an oversimplified example.

The scenario, as written by a product owner, is as follows:

Given I'm using the Temperature Converter.
When I enter 100 into Celsius field.
Then I obtain 212 in Fahrenheit field.

It would be translated into something similar to:

```java
@Given("I am using the Temperature Converter")
public void createTemperatureConverter() {
    // do nothing this is syntactic sugar for readability
}

@When("I enter \$celsius into Celsius field")
public void setCelsius(int celsius) {
    this.celsius = celsius;
}

@Then("I obtain \$fahrenheit in Fahrenheit field")
public void testCelsiusToFahrenheit(int fahrenheit) {
    assertEquals(fahrenheit,
        TemperatureConverter.celsiusToFahrenheit(celsius));
}
```

This allows both the programmers and the business users to speak the language of the domain (in this case, temperature conversions), and both are able to relate it back to their day-to-day work.

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

Android Studio and other IDE support

JUnit is fully supported by Android Studio, and it lets you create tested Android projects. Furthermore, you can run the tests and analyze the results without leaving the IDE (to some extent).

This also provides a more subtle advantage; being able to run the tests from the IDE allows you to debug the tests that are not behaving correctly.

In the following screenshot, we can see how ASide runs 19 unit tests, taking 1.043 seconds, with 0 Errors and 0 Failures detected. The name of each test and its duration is also displayed. If there were a failure, the Failure Trace would show the related information, as shown in the following screenshot:
There is also Android support in Eclipse IDE using the Android Development Tools plugin.

Even if you are not developing in an IDE, you can find support to run the tests with gradle (check http://gradle.org if you are not familiar with this tool). The tests are run using the command `gradle connectedAndroidTest`. This will install and run the tests for the debug build on a connected Android device.

This is actually the same method that Android Studio uses under the hood. ASide will just run the Gradle commands to build the project and run the tests, although with selective compilation.

### Java testing framework

The Java testing framework is the backbone of Android testing, and sometimes, you can get away without writing Android-specific code. This can be a good thing, because as we continue on our testing quest, you will notice that we deploy 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 filesystem content.

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.

Examples of things you can change are the test application package name, your test runner, or how to toggle performance-testing features:

```gradle
    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.

Then, the Instrumentation test runner is declared, which can be helpful if you create custom annotations to allow special behavior; for example, each test runs twice upon failure. In the case of not declaring a runner, the default custom runner android.test.InstrumentationTestRunner is used.
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. We will come back to this later.

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.

When you run a test application, the **Activity Manager** uses the instrumentation framework to start and control the test runner, which in turn uses instrumentation to shut down any running instances of the main application, starts the test application, and then starts the main application in the same process. This allows various aspects of the test application to work directly with the main application.

**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.

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.

Gradle and the Android Gradle plugin make for a powerful combination, and so, we will be using this build framework throughout the rest of the samples in this book.
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.

Every case has its pros and cons, but the good news is that you have all of these alternatives available to run your tests.

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.

The real devices are a requirement for performance tests, as it is somewhat difficult to extrapolate performance measurements from a simulated device. You will enjoy the real user experience only when using the real device. Rendering, scrolling, flinging, and other cases should be tested before delivering the application.

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)
Getting Started with Testing

The following screenshot shows the start of the form editor for reference:

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.

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.

Now that we have the basic infrastructure set up, it's time for us to start adding some tests, as shown in the following screenshot:

There's nothing to test right now, but as we are setting up the fundamentals of a Test-driven Development discipline, we are adding a dummy test just to get acquainted with the technique.

The src/androidTest/java folder in your AndroidApplicationTestingGuide project is the perfect place to add the tests. You could declare a different folder if you really wanted to, but we're sticking to defaults. The package should be the same as the corresponding package of the component being tested.

Right now, we are not concentrating on the content of the tests but on the concepts and placement of those tests.
Creating a test case

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 Test Case. Open the class under test (in this case, MainActivity) and hover over the class name until you see a lightbulb (or press Ctrl/Command + 1). Select Create Test from the menu that appears.

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

After entering all the required values, our JUnit test case creation dialog would look like this.

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.
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:

```java
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");
    }
}
```
The no-argument constructor is needed to run a specific test from the command line, as explained later using am 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.

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

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@SmallTest</td>
<td>Marks a test that should run as part of the small tests.</td>
</tr>
<tr>
<td>@MediumTest</td>
<td>Marks a test that should run as part of the medium tests.</td>
</tr>
<tr>
<td>@LargeTest</td>
<td>Marks a test that should run as part of the large tests.</td>
</tr>
<tr>
<td>@Smoke</td>
<td>Marks a test that should run as part of the smoke tests. The android.test.suitebuilder.SmokeTestSuiteBuilder will run all tests with this annotation.</td>
</tr>
<tr>
<td>@FlakyTest</td>
<td>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).</td>
</tr>
</tbody>
</table>
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:

```java
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.

Now that we have the tests in place, it's time to run them, and that's what we are going to do next.

Running the tests

There are several ways of running our tests, and we will analyze them here.

Additionally, as mentioned in the previous section about annotations, tests can be grouped or categorized and run together, depending on the situation.

Running all tests from Android Studio

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:

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 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:
Getting Started with Testing

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:

![Instrumentation screenshot](image)

When the tests are run in this way, the results can be seen through DDMS / LogCat, as described in the previous section.

**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.

The general usage of the am instrument command is:

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

This table summarizes the most common options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-r</code></td>
<td>Prints raw results. This is useful to collect raw performance data.</td>
</tr>
<tr>
<td><code>-e &lt;NAME&gt; &lt;VALUE&gt; </code></td>
<td>Sets arguments by name. We will examine its usage shortly. This is a generic option argument that allows us to set the <code>&lt;name, value&gt;</code> pairs.</td>
</tr>
<tr>
<td><code>-p &lt;FILE&gt;</code></td>
<td>Writes profiling data to an external file.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

To invoke the am command, we will be using the adb shell command or, if you already have a shell running on an emulator or device, you can issue the am command directly in the shell command prompt.

**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`

```java
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 tests from a specific test case**

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

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

**Running a specific test by name**

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

```bash
$: 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 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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-e unit true</code></td>
<td>This runs all unit tests. These are tests that are not derived from InstrumentationTestCase (and are not performance tests).</td>
</tr>
<tr>
<td><code>-e func true</code></td>
<td>This runs all functional tests. These are tests that are derived from InstrumentationTestCase.</td>
</tr>
<tr>
<td><code>-e perf true</code></td>
<td>This includes performance tests.</td>
</tr>
<tr>
<td>`-e size {small</td>
<td>medium</td>
</tr>
<tr>
<td><code>-e annotation &lt;annotation-name&gt;</code></td>
<td>This runs tests annotated with this annotation. This option is mutually exclusive with the size option.</td>
</tr>
</tbody>
</table>

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 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):

```java
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:

```java
@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 performance tests

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

Dry run

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. We won't be using this command; if you want more details, they can be found at [http://developer.android.com/reference/android/test/InstrumentationTestRunner.html](http://developer.android.com/reference/android/test/InstrumentationTestRunner.html).

### Other command-line options

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>true. Set break points in your code.</td>
</tr>
<tr>
<td>package</td>
<td>This is a fully qualified package name of one or several packages in the test application.</td>
</tr>
<tr>
<td>class</td>
<td>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 (#).</td>
</tr>
<tr>
<td>coverage</td>
<td>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 <em>Chapter 9, Alternative Testing Tactics</em>.</td>
</tr>
</tbody>
</table>
Summary
We have reviewed the main techniques and tools behind testing on Android. Having acquired this knowledge, it will let us begin our journey so that we can start exploiting the benefits of testing in our software development projects.

So far, we have visited the following subjects:

- We briefly analyzed the whys, whats, hows, and whens of testing. Henceforth, we will concentrate more on exploring the hows, now that you're giving testing the importance it deserves.
- We enumerated the different and most common types of tests you would need in your projects, described some of the tools we can count on our testing toolbox, and provided an introductory example of a JUnit unit test to better understand what we are discussing.
- We also created our first Android project with tests, using the Android Studio IDE and Gradle.
- We also created a simple test class to test the Activity in our project. We haven't added any useful test cases yet, but adding those simple ones was intended to validate our infrastructure.
- We also ran this simple test from our IDE and from the command line to understand the alternatives we have. In this process, we mentioned the Activity Manager and its command line incarnation am.
- We created a custom annotation to sort our tests and demonstrate how we can separate or differentiate suites of tests.

In the next chapter, we will start analyzing the mentioned techniques, frameworks, and tools in much greater detail, and provide examples of their usage.
Where to buy this book

You can buy Learning Android Application Testing from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.