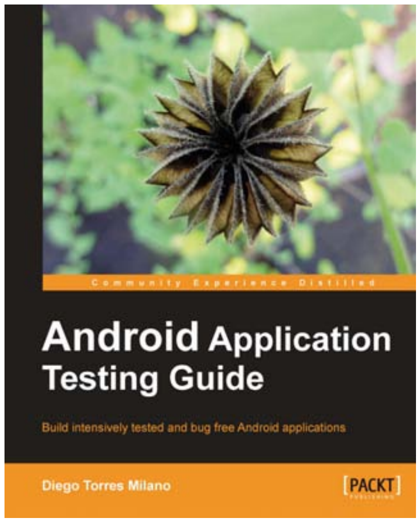
Android Application Testing Guide

Diego Torres Milano


Community Experience Distilled
Android Application Testing Guide
Build intensively tested and bug free Android applications
Diego Torres Milano
PACKT

Chapter No.1

"Getting Started with Testing"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Getting Started with Testing"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Diego Torres Milano has been involved with the Android platform since its inception, at the end of 2007, when he started exploring and researching the platform 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 in his personal blog (`http://dtmilano.blogspot.com`) and his participation as a lecturer in some conferences and courses like Mobile Dev Camp 2008 in Amsterdam (Netherlands) and Japan Linux Symposium 2009 (Tokyo), Droidcon London 2009, Skillsmatter 2009 (London, UK), and 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 (`cult-thinclient.sf.net`) and the very successful PXES Universal Linux Thin Client project (that was later acquired by 2X Software, `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 35M 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; organizations, institutions, and schools in some developing countries in South America, Africa, and Asia decided to use it because of the minimal hardware requirements to have a huge social impact providing computers, sometimes recycled ones, to everyone.

Among the other Open Source projects that he has founded we can mention Autoglade, Gnome-tla, JGlade, and he has been contributing to various Linux distributions such as RedHat, Fedora, and Ubuntu.

He also has been giving presentations in Linux World, LinuxTag, GUADEC ES, University of Buenos Aires, and so on.

He has been developing software, participating in Open Source projects, and advising companies worldwide for more than 15 years.

He can be contacted at `dtmilano@gmail.com`.

Firstly, I would like to thank my family: Laura, Augusto and Octavio for their patience and consideration. The time I borrowed to achieve this goal was mostly theirs.

Secondly I would like to thank my personal friend and IN3 Integracion Informatica co-founder, Caludio Palonsky, with whom we started this amazing adventure more than 15 years ago when we pioneered the provision of Linux services and support to enterprises in South America. He certainly taught me to be a bit more consultant and a bit less hacker (but I'm a very bad student :-)). And special thanks to Ricston's Peter Delia with whom we started providing Android training services throughout Europe as early as mid 2008 when Android was just a beautiful dream of having a mainstream Open Source operating system in the mobile arena. This is now a reality dictated by the market.

And lastly I would like to thank all the reviewers and the Packt Publishing team who gave me their opinion, suggestions, and corrections on early manuscripts; without them the book would never have had the quality it endowed.

Android Application Testing Guide

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 starts by introducing 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.

Chapter 2, Testing on Android covers testing on the Android platform, Unit testing and JUnit, creating an Android Test project, and running tests.

Chapter 3, Building Blocks on the Android SDK starts digging a bit deeper to recognize the building blocks available to create the tests. It covers Assertions, TouchUtils, intended to test User Interfaces, Mock objects, Instrumentation, and TestCase class hierarchies featuring UML diagrams.

Chapter 4, 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 5, 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 6, Behavior Driven Development introduces Behavior Driven Development and some concepts such as like the use of a common vocabulary to express the tests and the inclusion of business participants in the software development project.

Chapter 7, Testing Recipes provides practical examples of different common situations you will encounter 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, Local and Remote Services, UIs, Exceptions, Parsers, and Memory leaks.

Chapter 8, Continuous Integration introduces this agile technique for software engineering that aims to improve the software quality and to reduce the time taken to integrate changes by continuously applying integration and testing frequently.

Chapter 9, Performance Testing introduces a series of concepts related to benchmarking and profiles from traditional logging statement methods to Creating Android performance tests and using profiling tools. This chapter also presents Caliper to create microbenchmarks.

Chapter 10, Alternative Testing Tactics covers building Android from source, code coverage using EMMA, Robotium, testing on hosts, and Robolectric.

1
Getting Started with Testing

This chapter introduces the different types of testing and their applicability to software development projects in general and to **Android** in particular.

We will avoid introductions to Android and the **Open Handset Alliance** (`http://www.openhandsetalliance.com`) as they are covered in many books already and I am inclined to believe that if you are reading a book covering this more advanced topic you will have started with Android development before.

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

Brief history

Initially, when Android was introduced by the end of 2007, there was very little support for testing on the platform, and for some of us very accustomed to using testing as a component intimately coupled with the development process, it was time to start developing some frameworks and tools to permit this approach.

By that time Android had some rudimentary support for unit testing using JUnit (`http://www.JUnit.org`), but it was not fully supported and even less documented.

In the process of writing my own library and tools, I discovered Phil Smith's **Positron** (originally at `http://code.google.com/p/android-positron` and now renamed and moved to `http://code.google.com/p/autoandroid`), an Open Source library and a very suitable alternative to support testing on Android, so I decided to extend his excellent work and bring some new and missing pieces to the table.

Some aspects of test automation were not included and I started a complementary project to fill that gap, it was consequently named **Electron**. And although positron is the anti-particle of the electron, and they annihilate if they collide, take for granted that that was not the idea, but more the conservation of energy and the generation of some visible light and waves.

Later on, Electron entered the first **Android Development Challenge** (**ADC1**) in early 2008 and though it obtained a rather good score in some categories, frameworks had no place in that competition. Should you be interested in the origin of testing on Android, please find some articles and videos that were published in my personal blog (`http://dtmilano.blogspot.com/search/label/electron`).

By that time Unit Tests could be run on Eclipse. However, testing was not done on the real target but on a JVM on the local development computer.

Google also provided application instrumentation code through the `Instrumentation` class. When running an application with instrumentation turned on, this class is instantiated for you before any of the application code, allowing you to monitor all of the interaction the system has with the application. An Instrumentation implementation is described to the system through an `AndroidManifest.xml` file.

During those early stages in the Android development evolution, I started writing some articles in my blog filling the gaps on testing. This book is the evolution and completion of that work in an orderly and understandable manner to paradoxically let you be bitten by the Android testing bug.

Software bugs

It doesn't matter how hard you try and how much time you invest in design and even how careful you are when programming, mistakes are inevitable and bugs will appear.

Bugs and software development are intimately related. However, the term **bugs** to describe flaws, mistakes, or errors has been used in hardware engineering many decades before even computers were invented. Notwithstanding the story about the term 'bug' coined by Mark II operators at Harvard University, Thomas Edison wrote this in 1878 in a letter to Puskás Tivadar showing the early adoption of the term:

> *"It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise – this thing gives out and [it is] then that 'Bugs' – as such little faults and difficulties are called – show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached."*

How bugs severely affect your projects

Bugs affect many aspects of your software development project and it is clearly understood that the sooner in the process you find and *squash* them, the better. It doesn't matter if you are developing a simple application to publish on the Android Market, re-branding the Android experience for an operator, or creating a customized version of Android for a device manufacturer, bugs will delay your shipment and will cost you money.

From all of the software development methodologies and techniques, **Test Driven Development**, an agile component of the software development process, is likely the one that forces you to face your bugs earlier in the development process and thus it is also likely that you will solve more problems up front.

Furthermore, the increase in productivity can be clearly appreciated in a project where a software development team uses this technique versus one that is, in the best of cases, writing tests at the end of the development cycle. If you have been involved in software development for the mobile industry, you will have reasons to believe that with all the rush this stage never occurs. It's funny because, usually, this rush is to solve problems that could have been avoided.

In a study conducted by the **National Institute of Standards and Technology** (USA) in 2002, it was reported that software bugs cost the economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed.

But please, don't misunderstand this message. There are no *silver bullets* in software development and what will lead you to an increase in productivity and manageability of your project is discipline in applying these methodologies and techniques to stay in control.

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 software tests for your development project. Increased productivity will soon be evident.

Additionally, writing the 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 ability to confidently change or update it.

The more the code covered by your tests, the higher would be your expectations of discovering the 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.

This technique requires a special instrumented Android build to collect probe data and must be disabled for any release code because the impact on performance could severely affect application behavior.

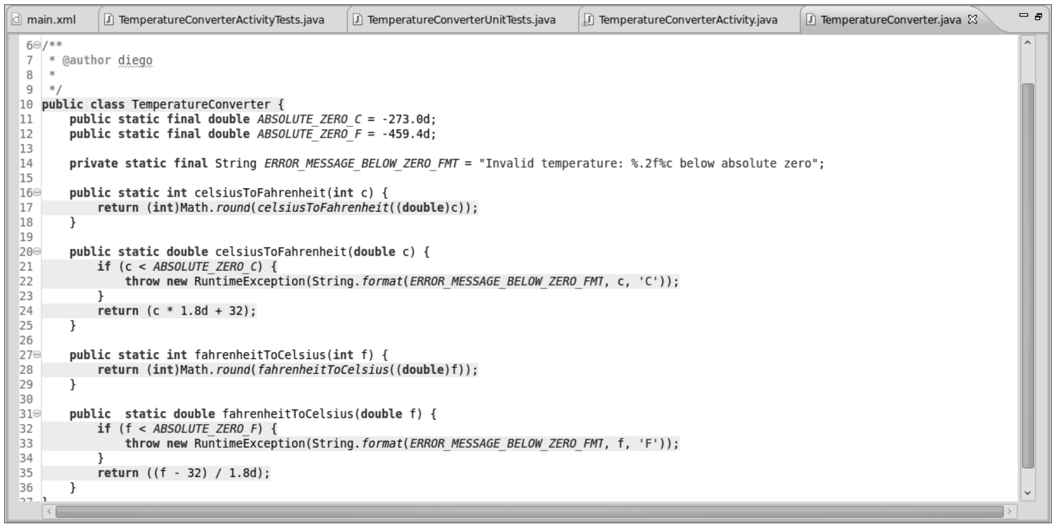
To fill this gap, enter EMMA (`http://emma.sourceforge.net/`), an open-source toolkit for measuring and reporting Java code coverage, that can offline instrument classes for coverage. It supports various coverage types:

- class
- method
- line
- basic block

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

We will be analyzing the use of EMMA on Android to guide us to full test coverage of our code in *Chapter 10*, *Alternative Testing Tactics*.

This screenshot shows how an EMMA code coverage report is displayed in the Eclipse editor, showing green lines where the code has been tested, provided the corresponding plugin is installed.


main.xml
TemperatureConverterActivityTests.java
TemperatureConverterUnitTests.java
TemperatureConverterActivity.java
TemperatureConverter.java

```java
/**
 * @author diego
 *
 */
public class TemperatureConverter {
    public static final double ABSOLUTE_ZERO_C = -273.0d;
    public static final double ABSOLUTE_ZERO_F = -459.4d;

    private static final String ERROR_MESSAGE_BELOW_ZERO_FMT = "Invalid temperature: %.2f%c below absolute zero";

    public static int celsiusToFahrenheit(int c) {
        return (int)Math.round(celsiusToFahrenheit((double)c));
    }

    public static double celsiusToFahrenheit(double c) {
        if (c < ABSOLUTE_ZERO_C) {
            throw new RuntimeException(String.format(ERROR_MESSAGE_BELOW_ZERO_FMT, c, 'C'));
        }
        return (c * 1.8d + 32);
    }

    public static int fahrenheitToCelsius(int f) {
        return (int)Math.round(fahrenheitToCelsius((double)f));
    }

    public  static double fahrenheitToCelsius(double f) {
        if (f < ABSOLUTE_ZERO_F) {
            throw new RuntimeException(String.format(ERROR_MESSAGE_BELOW_ZERO_FMT, f, 'F'));
        }
        return ((f - 32) / 1.8d);
    }
```

Unfortunately, the plugin doesn't support Android tests yet, so right now you can only use it for your JUnit tests. An Android coverage analysis report is only available through HTML.

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

This leads us to the introduction of **Continuous Integration**, which will be discussed in detail in *Chapter 8*, *Continuous Integration*. This relies on the automation of tests and building processes.

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 do not break existing code.

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 test the path of execution or just some methods. Usually there is 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 own tests.

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

Activity lifecycle events

You should test that your activities handle lifecycle events correctly.

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

Configuration-changed events should also be tested as some of these events cause the current Activity to be recreated, and you should test for correct handling of the event and that the newly created Activity preserves the previous state. Configuration changes are triggered even by rotation events, 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 they are handled correctly. These operations should be tested in isolation at the lower system level, at a higher level through `ContentProviders`, and from the application itself.

To test these components in isolation, Android provides some mock objects in the `android.test.mock` package.

Physical characteristics of the device

Well before delivering your application you should be sure that all of the different devices it can be run on are supported or at the least you should detect the situation and take appropriate measures.

Among other characteristics of the devices, you may find that you should test:

- Network capabilities
- Screen densities
- Screen resolutions
- Screen sizes
- Availability of sensors

- Keyboard and other input devices
- GPS
- External storage

In this respect Android Virtual Devices play an important role because it is practically impossible to have access to all possible devices with all of the possible combinations of features but you can configure AVD for almost every situation. However, as was mentioned before, save your final testing for actual devices where real users will run the application to understand its behavior.

Types of tests

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

There are several types of test available depending on the object being tested. Regardless of its type, a test should verify a condition and return the result of this evaluation as a single Boolean value indicating success or failure.

Unit tests

Unit tests are software tests written by programmers for programmers in a programming language and they should isolate the component under test 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 not found the next time the test is run.

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 (up to Android 2.3 Gingerbread) uses JUnit 3. This version doesn't use annotations and uses introspection to detect the tests.

A typical JUnit test would look something like this (the actual tests are highlighted):

```
/**
 * Android Application Testing Guide
 */
package com.example.aatg.test;
```

```
import JUnit.framework.TestCase;
/**
 * @author diego
 */
public class MyUnitTests extends TestCase {
  private int mFixture;
/**
 * @param name test name
 */
  public MyUnitTests(String name) {
    super(name);
  }
/* (non-Javadoc)
 * @see JUnit.framework.TestCase#setUp()
 */
  protected void setUp() throws Exception {
    super.setUp();
    mFixture = 1234;
  }
/* (non-Javadoc)
 * @see JUnit.framework.TestCase#tearDown()
 */
  protected void tearDown() throws Exception {
    super.tearDown();
  }
/**
 * Preconditions
 */
  public void testPreconditions() {
  }
/**
 * Test method
 */
  public void testSomething() {
    fail("Not implemented yet");
  }
}
```

Downloading the example code

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 in detail the components that build up our test case.

The test fixture

A test fixture is the well known state defined as a baseline to run the tests and is shared by all the test cases, and thus plays a fundamental role in the design of the tests.

Generally it is implemented as a set of member variables and, following Android conventions, they will have names starting with `m`, for example `mActivity`. However, it can also contain external data, as specific entries in a database or files present in the filesystem.

The setUp() method

This method is called to initialize the fixture.

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 a network connection here.

JUnit is designed in such 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 end of the entire test run. This is particularly important in Android and when testing on limited devices as some tests may fail not because of an intrinsic problem but because of the amount of memory needed to run the application plus 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.

Test preconditions

Usually there is no way to test for preconditions as the tests are discovered using introspection and their order could vary. So it's customary to create a `testPreconditions()` method to test for preconditions. Though there is no assurance that this test will be called in any specific order, it is good practice to keep this and the preconditions together for organizational purposes.

The actual tests

All `public void` methods whose names start with `test` will be considered as a test. JUnit 3, as opposed to JUnit 4, doesn't use annotations to discover the tests but introspection to find their names. There are some annotations available on the Android test framework such as `@SmallTest`, `@MediumTest`, and `@LargeTest`, but they don't turn a simple method into a test. Instead they 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 using nouns and the condition being tested.

For example: `testValues()`, `testConversionError()`, `testConversionToString()` are all valid test names.

Test for exceptions and wrong values instead of just testing for positive cases.

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 with them throwing exceptions if 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:

- `assertEquals()`
- `assertFalse()`
- `assertNotNull()`
- `assertNotSame()`
- `assertNull()`
- `assertSame()`
- `assertTrue()`
- `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 done to ensure that correct methods are called but they can also be of help, as mentioned, to isolate your tests from the surrounding universe and enable you to run them independently and repeatably.

The Android testing framework supports several mock objects that you will find very useful when writing your tests but you will need to provide some dependencies to be able to compile the tests.

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 where every method generates an `UnsupportedOperationException` and that you can extend to create real mock objects.

UI tests

Finally, special consideration 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 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 providing the corresponding `Runnable` containing testing instructions.

A helper class `TouchUtils` is also provided to aid in the UI test creation allowing the generation of events to send to the Views, such as:

- click
- drag
- long click
- scroll
- tap
- touch

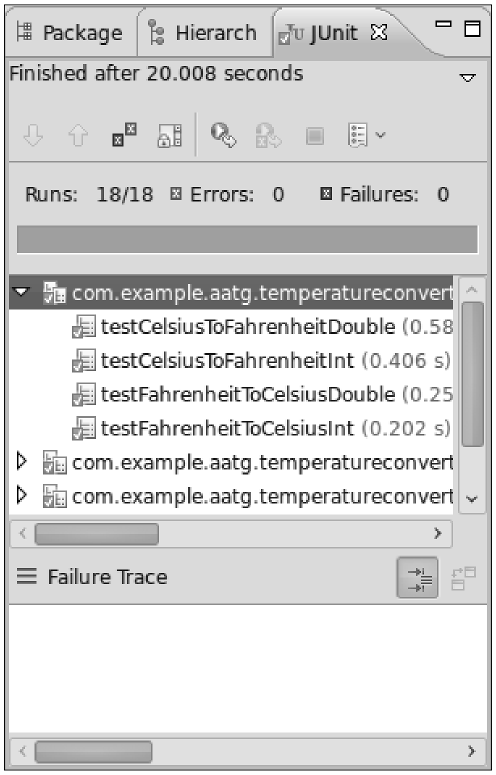
By these means you can actually remote control you application from the tests.

Eclipse and other IDE support

JUnit is fully supported by Eclipse and the Android ADT plugin lets you create Android testing projects. Furthermore, you can run the tests and analyze the results without leaving the IDE.

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

In the screenshot, we can see how Eclipse runs **18 tests** taking 20.008 seconds, where **0 Errors** and **0 Failure**s were detected. The name of each test and its duration is also displayed. If there was a failure, the **Failure Trace** would show the related information.



Finished after 20.008 seconds

Runs: 18/18 Errors: 0 Failures: 0

```
com.example.aatg.temperatureconvert
    testCelsiusToFahrenheitDouble (0.58
    testCelsiusToFahrenheitInt (0.406 s)
    testFahrenheitToCelsiusDouble (0.25
    testFahrenheitToCelsiusInt (0.202 s)
com.example.aatg.temperatureconvert
com.example.aatg.temperatureconvert
```

Other IDEs like ItelliJ and Netbeans have plugins integrating Android development to some degree but they are not officially supported.

Even if you are not developing in an IDE, you can find support to run the tests with **ant** (check `http://ant.apache.org` if you are not familiar with this tool). This setup is done by the `android` command using the subcommand `create test-project` as described by this help text:

```
$ android --help create test-project

Usage:
  android [global options] create test-project [action options]

Global options:
  -v --verbose  Verbose mode: errors, warnings and informational messages
are printed.
  -h --help     Help on a specific command.
  -s --silent   Silent mode: only errors are printed out.
```

```
Action "create test-project":
  Creates a new Android project for a test package.
Options:
  -p --path     The new project's directory [required]
  -m --main     Path to directory of the app under test, relative to the
test project directory [required]
  -n --name     Project name
```

As indicated by the help you should provide at least the path to the project (`--path`) and the path to the main project or the project under test (`--main`).

Integration tests

Integration tests are designed to test the way individual components work jointly. 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, filesystem, and databases.

The same criteria apply to other Android components like `Services` or `ContentProviders` that need to interact with other parts of the system to achieve their function.

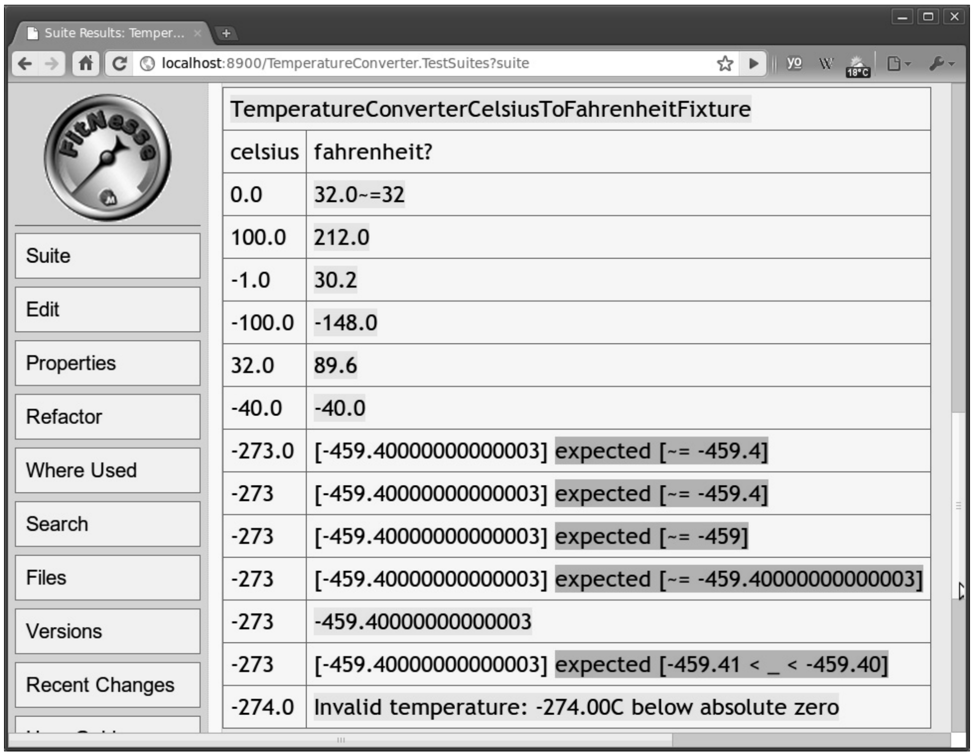
In all these cases there are specialized tests provided by the Android testing framework that facilitate the creation of tests for these components.

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 test the completeness and correctness of a user requirement 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, most notably FitNesse (`http://www.fitnesse.org`), which can be easily integrated, up to a point, into the Android development process and will let you create acceptance tests and check their results.

Also check Fit, `http://fit.c2.com` and Slim (Simple List Invocation Method), `http://fitnesse.org/FitNesse.UserGuide.SliM`, as an alternative to Fit.



TemperatureConverterCelsiusToFahrenheitFixture	
celsius	fahrenheit?
0.0	32.0~=32
100.0	212.0
-1.0	30.2
-100.0	-148.0
32.0	89.6
-40.0	-40.0
-273.0	[-459.40000000000003] expected [~= -459.4]
-273	[-459.40000000000003] expected [~= -459.4]
-273	[-459.40000000000003] expected [~= -459]
-273	[-459.40000000000003] expected [~= -459.40000000000003]
-273	-459.40000000000003
-273	[-459.40000000000003] expected [-459.41 < _ < -459.40]
-274.0	Invalid temperature: -274.00C below absolute zero

Lately, a new trend named **Behavior Driven Development** has gained some popularity and in a very brief description can be understood as the evolution 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`):

- 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 some tool, such as **jbehave** (`http://jbehave.org`). In the following example, these scenarios are translated into code that expresses the same test scenario in a programming language.

Test case scenario

As an illustration of this technique here is an oversimplified example.

This scenario is:

```
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:

```
@Given("I am using the Temperature Converter")
public void createTemperatureConverter() {
    // do nothing
}
@When("I enter $celsius into Celsius field")
public void setCelsius(int celsius) {
    mCelsius= celsius;
}
@Then("I obtain $fahrenheit in Fahrenheit field")
public void testCelsiusToFahrenheit(int fahrenheit) {
    assertEquals(fahrenheit,
      TemperatureConverter.celsiusToFahrenheit
        (mCelsius));
}
```

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 some 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 on 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 like:

- GUI tests
- Smoke tests
- Performance tests
- Installation tests

Android testing framework

Android provides a very advanced testing framework extending the industry standard JUnit with specific features suitable for implementing 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 cases simple and straightforward.

The 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 tests control and examine the application.
- Mock versions of commonly used Android system objects.
- Tools for running single tests or test suites, with or without instrumentation.
- Support for managing tests and test projects in the ADT Plugin for Eclipse and at the command line.

Instrumentation

The instrumentation framework is the foundation of the testing framework. Instrumentation controls the application under test 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 them.

All interaction 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, forcing the values returned by some methods or mocking persistent and unchanged data for `ContentProvider`, databases, or even the filesystem content.

A standard Android project has its tests in a correlated project that usually has the same project name but ends with **Test**. Inside this Test project, the `AndroidManifest.xml` declares the Instrumentation.

As an illustrative example, assume your project has a manifest like this:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.aatg.sample"
      android:versionCode="1"
      android:versionName="1.0">
      <application android:icon="@drawable/icon"
             android:label="@string/app_name">
        <activity android:name=".SampleActivity"
                  android:label="@string/app_name">
          <intent-filter>
             <action android:name="android.intent.action.MAIN" />
             <category android:name=
               "android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
      </application>
    <uses-sdk android:minSdkVersion="7" />
</manifest>
```

In this case, the correlated Test project will have the following `AndroidManifest.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.aatg.sample.test"
  android:versionCode="1" android:versionName="1.0">
  <application android:icon="@drawable/icon"
              android:label="@string/app_name">
    <uses-library android:name="android.test.runner" />
  </application>
  <uses-sdk android:minSdkVersion="7" />
  <instrumentation
    android:targetPackage="com.example.aatg.sample"
```

```
        android:name="android.test.InstrumentationTestRunner"
        android:label="Sample Tests" />
      <uses-permission android:name="
        android.permission.INJECT_EVENTS" />
    </manifest>
```

Here the Instrumentation package is the same package as the main application with the `.test` suffix added.

Then the Instrumentation is declared specifying the target package and the test runner, in this case the default custom runner `android.test.InstrumentationTestRunner`.

Also notice that both, the application under test and the tests 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** (`http://developer.android.com/intl/de/reference/android/app/ActivityManager.html`) 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.

Test targets

During the evolution of your development project your tests would be targeted to different devices. From the simplicity, flexibility, and speed of testing on an emulator to the unavoidable final testing on the specific devices you intend your application to be run on, you should be able to run on all of them.

There are also some intermediate cases like 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 is 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 discover the real user experience only when using the real device. Rendering, scrolling, flinging, and other cases should be tested before delivering the application.

Summary

We have reviewed the main concepts behind testing in general and Android in particular. Having acquired this knowledge will let us start our journey and start exploiting the benefits of testing in our software development projects.

So far, we have visited the following subjects:

- We reviewed the early stages of testing on Android and mentioned some of the frameworks that created the current alternatives.
- We briefly analyzed the reasons behind testing and the whys, whats, hows, and whens of it. Furthermore, from now on we will concentrate on exploring the hows, as we can assume that you are convinced by the arguments presented.
- 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 analyzed these techniques from the Android perspective and mentioned the use of Instrumentation to run our Android tests.

Now we will start analyzing the mentioned techniques, frameworks, and tools in detail, along with examples of their usage.