Testing Fundamentals

The Android testing framework, an integral part of the development environment, provides an architecture and powerful tools that help you test every aspect of your application at every level from unit to framework.

The testing framework has these key features:

- Android test suites are based on JUnit. You can use plain JUnit to test a class that doesn't call the Android API, or Android's JUnit extensions to test Android components. If you're new to Android testing, you can start with general-purpose test case classes such as `AndroidTestCase` and then go on to use more sophisticated classes.
- The Android JUnit extensions provide component-specific test case classes. These classes provide helper methods for creating mock objects and methods that help you control the lifecycle of a component.
- Test suites are contained in test packages that are similar to main application packages, so you don't need to learn a new set of tools or techniques for designing and building tests.
- The SDK tools for building and tests are available in Eclipse with ADT, and also in command-line form for use with other IDEs. These tools get information from the project of the application under test and use this information to automatically create the build files, manifest file, and directory structure for the test package.
- The SDK also provides monkeyrunner, an API for testing devices with Python programs, and UI/Application Exerciser Monkey, a command-line tool for stress-testing UIs by sending pseudo-random events to a device.

This document describes the fundamentals of the Android testing framework, including the structure of tests, the APIs that you use to develop tests, and the tools that you use to run tests and view results. The document assumes you have a basic knowledge of Android application programming and JUnit testing methodology.

The following diagram summarizes the testing framework:

IN THIS DOCUMENT

KEY CLASSES

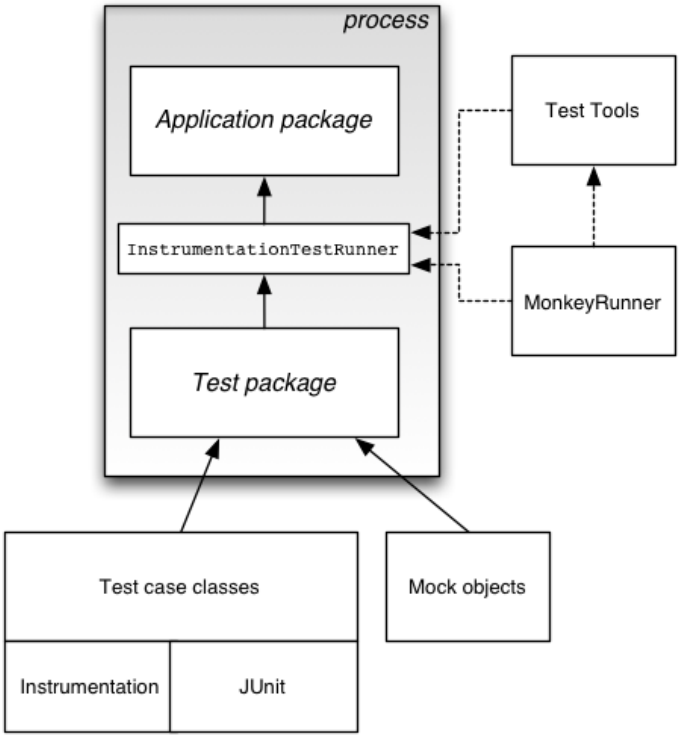
- `InstrumentationTestRunner`
- `android.test`
- `android.test.mock`
- `junit.framework`

RELATED TUTORIALS

- Activity Testing Tutorial

SEE ALSO

- Testing from Eclipse with ADT
- Testing from Other IDEs
- monkeyrunner
- UI/Application Exerciser Monkey



Test Structure

Android's build and test tools assume that test projects are organized into a standard structure of tests, test case classes, test packages, and test projects.

Android testing is based on JUnit. In general, a JUnit test is a method whose statements test a part of the application under test. You organize test methods into classes called test cases (or test suites). Each test is an isolated test of an individual module in the application under test. Each class is a container for related test methods, although it often provides helper methods as well.

In JUnit, you build one or more test source files into a class file. Similarly, in Android you use the SDK's build tools to build one or more test source files into class files in an Android test package. In JUnit, you use a test runner to execute test classes. In Android, you use test tools to load the test package and the application under test, and the tools then execute an Android-specific test runner.

Test Projects

Tests, like Android applications, are organized into projects.

A test project is a directory or Eclipse project in which you create the source code, manifest file, and other files for a test package. The Android SDK contains tools for Eclipse with ADT and for the command line that create and update test projects for you. The tools create the directories you use for source code and resources and the manifest file for the test package. The command-line tools also create the Ant build files you need.

You should always use Android tools to create a test project. Among other benefits, the tools:

- Automatically set up your test package to use `InstrumentationTestRunner` as the test case runner. You must use `InstrumentationTestRunner` (or a subclass) to run JUnit tests.
- Create an appropriate name for the test package. If the application under test has a package name of `com.mydomain.myapp`, then the Android tools set the test package name to `com.mydomain.myapp.test`. This helps you identify their relationship, while preventing conflicts within the system.
- Automatically create the proper build files, manifest file, and directory structure for the test project. This helps you to build the test package without having to modify build files and sets up the linkage between your test package and the application under test. The

You can create a test project anywhere in your file system, but the best approach is to add the test project so that its root directory `tests/` is at the same level as the `src/` directory of the main application's project. This helps you find the tests

associated with an application. For example, if your application project's root directory is `MyProject`, then you should use the following directory structure:

```
MyProject/
    AndroidManifest.xml
    res/
        ... (resources for main application)
    src/
        ... (source code for main application) ...
    tests/
        AndroidManifest.xml
        res/
            ... (resources for tests)
        src/
            ... (source code for tests)
```

The Testing API

The Android testing API is based on the JUnit API and extended with a instrumentation framework and Android-specific testing classes.

JUnit

You can use the JUnit `TestCase (/reference/junit/framework/TestCase.html)` class to do unit testing on a class that doesn't call Android APIs. `TestCase` is also the base class for `AndroidTestCase (/reference/android/test/AndroidTestCase.html)`, which you can use to test Android-dependent objects. Besides providing the JUnit framework, AndroidTestCase offers Android-specific setup, teardown, and helper methods.

You use the JUnit `Assert (/reference/junit/framework/Assert.html)` class to display test results. The assert methods compare values you expect from a test to the actual results and throw an exception if the comparison fails. Android also provides a class of assertions that extend the possible types of comparisons, and another class of assertions for testing the UI. These are described in more detail in the section Assertion classes (#AssertionClasses)

To learn more about JUnit, you can read the documentation on the junit.org (http://www.junit.org) home page. Note that the Android testing API supports JUnit 3 code style, but not JUnit 4. Also, you must use Android's instrumented test runner `InstrumentationTestRunner (/reference/android/test/InstrumentationTestRunner.html)` to run your test case classes. This test runner is described in the section Running Tests (#InstrumentationTestRunner).

Instrumentation

Android instrumentation is a set of control methods or "hooks" in the Android system. These hooks control an Android component independently of its normal lifecycle. They also control how Android loads applications.

Normally, an Android component runs in a lifecycle determined by the system. For example, an Activity object's lifecycle starts when the Activity is activated by an Intent. The object's `onCreate()` method is called, followed by `onResume()`. When the user starts another application, the `onPause()` method is called. If the Activity code calls the `finish()` method, the `onDestroy()` method is called. The Android framework API does not provide a way for your code to invoke these callback methods directly, but you can do so using instrumentation.

Also, the system runs all the components of an application into the same process. You can allow some components, such as content providers, to run in a separate process, but you can't force an application to run in the same process as another application that is already running.

With Android instrumentation, though, you can invoke callback methods in your test code. This allows you to run through the lifecycle of a component step by step, as if you were debugging the component. The following test code snippet demonstrates how to use this to test that an Activity saves and restores its state:

```
    // Start the main activity of the application under test
    mActivity = getActivity();

    // Get a handle to the Activity object's main UI widget, a Spinner
    mSpinner = (Spinner)mActivity.findViewById(com.android.example.spinner.R.id.Spinner01);

    // Set the Spinner to a known position
    mActivity.setSpinnerPosition(TEST_STATE_DESTROY_POSITION);

    // Stop the activity - The onDestroy() method should save the state of the Spinner
```

```
    mActivity.finish();

    // Re-start the Activity - the onResume() method should restore the state of the Spinner
    mActivity = getActivity();

    // Get the Spinner's current position
    int currentPosition = mActivity.getSpinnerPosition();

    // Assert that the current position is the same as the starting position
    assertEquals(TEST_STATE_DESTROY_POSITION, currentPosition);
```

The key method used here is `getActivity()` (/reference/android/test/ActivityInstrumentationTestCase2.html#getActivity()), which is a part of the instrumentation API. The Activity under test is not started until you call this method. You can set up the test fixture in advance, and then call this method to start the Activity.

Also, instrumentation can load both a test package and the application under test into the same process. Since the application components and their tests are in the same process, the tests can invoke methods in the components, and modify and examine fields in the components.

Test case classes

Android provides several test case classes that extend `TestCase` (/reference/junit/framework/TestCase.html) and `Assert` (/reference/junit/framework/Assert.html) with Android-specific setup, teardown, and helper methods.

AndroidTestCase

A useful general test case class, especially if you are just starting out with Android testing, is `AndroidTestCase` (/reference/android/test/AndroidTestCase.html). It extends both `TestCase` (/reference/junit/framework/TestCase.html) and `Assert` (/reference/junit/framework/Assert.html). It provides the JUnit-standard `setUp()` and `tearDown()` methods, as well as all of JUnit's Assert methods. In addition, it provides methods for testing permissions, and a method that guards against memory leaks by clearing out certain class references.

Component-specific test cases

A key feature of the Android testing framework is its component-specific test case classes. These address specific component testing needs with methods for fixture setup and teardown and component lifecycle control. They also provide methods for setting up mock objects. These classes are described in the component-specific testing topics:

- Activity Testing
- Content Provider Testing
- Service Testing

Android does not provide a separate test case class for BroadcastReceiver. Instead, test a BroadcastReceiver by testing the component that sends it Intent objects, to verify that the BroadcastReceiver responds correctly.

ApplicationTestCase

You use the `ApplicationTestCase` (/reference/android/test/ApplicationTestCase.html) test case class to test the setup and teardown of `Application` (/reference/android/app/Application.html) objects. These objects maintain the global state of information that applies to all the components in an application package. The test case can be useful in verifying that the <application> element in the manifest file is correctly set up. Note, however, that this test case does not allow you to control testing of the components within your application package.

InstrumentationTestCase

If you want to use instrumentation methods in a test case class, you must use `InstrumentationTestCase` (/reference/android/test/InstrumentationTestCase.html) or one of its subclasses. The `Activity` (/reference/android/app/Activity.html) test cases extend this base class with other functionality that assists in Activity testing.

Assertion classes

Because Android test case classes extend JUnit, you can use assertion methods to display the results of tests. An assertion method compares an actual value returned by a test to an expected value, and throws an AssertionException if the comparison test fails. Using assertions is more convenient than doing logging, and provides better test performance.

Besides the JUnit Assert (/reference/junit/framework/Assert.html) class methods, the testing API also provides the MoreAsserts (/reference/android/test/MoreAsserts.html) and ViewAsserts (/reference/android/test/ViewAsserts.html) classes:

- MoreAsserts contains more powerful assertions such as assertContainsRegex(String, String), which does regular expression matching.
- ViewAsserts contains useful assertions about Views. For example it contains assertHasScreenCoordinates(View, View, int, int) that tests if a View has a particular X and Y position on the visible screen. These asserts simplify testing of geometry and alignment in the UI.

Mock object classes

To facilitate dependency injection in testing, Android provides classes that create mock system objects such as Context (/reference/android/content/Context.html) objects, ContentProvider (/reference/android/content/ContentProvider.html) objects, ContentResolver (/reference/android/content/ContentResolver.html) objects, and Service (/reference/android/app/Service.html) objects. Some test cases also provide mock Intent (/reference/android/content/Intent.html) objects. You use these mocks both to isolate tests from the rest of the system and to facilitate dependency injection for testing. These classes are found in the packages android.test (/reference/android/test/package-summary.html) and android.test.mock (/reference/android/test/mock/package-summary.html).

Mock objects isolate tests from a running system by stubbing out or overriding normal operations. For example, a MockContentResolver (/reference/android/test/mock/MockContentResolver.html) replaces the normal resolver framework with its own local framework, which is isolated from the rest of the system. MockContentResolver also stubs out the notifyChange(Uri, ContentObserver, boolean) (/reference/android/content/ContentResolver.html#notifyChange(android.net.Uri, android.database.ContentObserver, boolean)) method so that observer objects outside the test environment are not accidentally triggered.

Mock object classes also facilitate dependency injection by providing a subclass of the normal object that is non-functional except for overrides you define. For example, the MockResources (/reference/android/test/mock/MockResources.html) object provides a subclass of Resources (/reference/android/content/res/Resources.html) in which all the methods throw Exceptions when called. To use it, you override only those methods that must provide information.

These are the mock object classes available in Android:

Simple mock object classes

MockApplication (/reference/android/test/mock/MockApplication.html), MockContext (/reference/android/test/mock/MockContext.html), MockContentProvider (/reference/android/test/mock/MockContentProvider.html), MockCursor (/reference/android/test/mock/MockCursor.html), MockDialogInterface (/reference/android/test/mock/MockDialogInterface.html), MockPackageManager (/reference/android/test/mock/MockPackageManager.html), and MockResources (/reference/android/test/mock/MockResources.html) provide a simple and useful mock strategy. They are stubbed-out versions of the corresponding system object class, and all of their methods throw an UnsupportedOperationException (/reference/java/lang/UnsupportedOperationException.html) exception if called. To use them, you override the methods you need in order to provide mock dependencies.

Note: MockContentProvider (/reference/android/test/mock/MockContentProvider.html) and MockCursor (/reference/android/test/mock/MockCursor.html) are new as of API level 8.

Resolver mock objects

MockContentResolver (/reference/android/test/mock/MockContentResolver.html) provides isolated testing of content providers by masking out the normal system resolver framework. Instead of looking in the system to find a content provider given an authority string, MockContentResolver uses its own internal table. You must explicitly add providers to this table using addProvider(String, ContentProvider) (/reference/android/test/mock/MockContentResolver.html#addProvider(java.lang.String, android.content.ContentProvider)).

With this feature, you can associate a mock content provider with an authority. You can create an instance of a real provider but use test data in it. You can even set the provider for an authority to `null`. In effect, a MockContentResolver object isolates your test from providers that contain real data. You can control the function of the provider, and you can prevent your test from affecting real data.

Contexts for testing

Android provides two Context classes that are useful for testing:

- `IsolatedContext` provides an isolated `Context`, File, directory, and database operations that use this Context take place in a test area. Though its functionality is limited, this Context has enough stub code to respond to system calls.

  This class allows you to test an application's data operations without affecting real data that may be present on the device.

- `RenamingDelegatingContext` provides a Context in which most functions are handled by an existing `Context`, but file and database operations are handled by a `IsolatedContext`. The isolated part uses a test directory and creates special file and directory names. You can control the naming yourself, or let the constructor determine it automatically.

  This object provides a quick way to set up an isolated area for data operations, while keeping normal functionality for all other Context operations.

Running Tests

Test cases are run by a test runner class that loads the test case class, set ups, runs, and tears down each test. An Android test runner must also be instrumented, so that the system utility for starting applications can control how the test package loads test cases and the application under test. You tell the Android platform which instrumented test runner to use by setting a value in the test package's manifest file.

`InstrumentationTestRunner (/reference/android/test/InstrumentationTestRunner.html)` is the primary Android test runner class. It extends the JUnit test runner framework and is also instrumented. It can run any of the test case classes provided by Android and supports all possible types of testing.

You specify `InstrumentationTestRunner` or a subclass in your test package's manifest file, in the `<instrumentation> (/guide/topics/manifest/instrumentation-element.html)` element. Also, `InstrumentationTestRunner` code resides in the shared library `android.test.runner`, which is not normally linked to Android code. To include it, you must specify it in a `<uses-library> (/guide/topics/manifest/uses-library-element.html)` element. You do not have to set up these elements yourself. Both Eclipse with ADT and the `android` command-line tool construct them automatically and add them to your test package's manifest file.

Note: If you use a test runner other than `InstrumentationTestRunner`, you must change the <instrumentation> element to point to the class you want to use.

To run `InstrumentationTestRunner (/reference/android/test/InstrumentationTestRunner.html)`, you use internal system classes called by Android tools. When you run a test in Eclipse with ADT, the classes are called automatically. When you run a test from the command line, you run these classes with Android Debug Bridge (adb) (/tools/help/adb.html).

The system classes load and start the test package, kill any processes that are running an instance of the application under test, and then load a new instance of the application under test. They then pass control to `InstrumentationTestRunner (/reference/android/test/InstrumentationTestRunner.html)`, which runs each test case class in the test package. You can also control which test cases and methods are run using settings in Eclipse with ADT, or using flags with the command-line tools.

Neither the system classes nor `InstrumentationTestRunner (/reference/android/test/InstrumentationTestRunner.html)` run the application under test. Instead, the test case does this directly. It either calls methods in the application under test, or it calls its own methods that trigger lifecycle events in the application under test. The application is under the complete control of the test case, which allows it to set up the test environment (the test fixture) before running a test. This is demonstrated in the previous code snippet (#ActivitySnippet) that tests an Activity that displays a Spinner widget.

To learn more about running tests, please read the topics Testing from Eclipse with ADT (/tools/testing/testing_eclipse.html) or Testing from Other IDEs (/tools/testing/testing_otheride.html).

Seeing Test Results

The Android testing framework returns test results back to the tool that started the test. If you run a test in Eclipse with ADT, the results are displayed in a new JUnit view pane. If you run a test from the command line, the results are displayed in `STDOUT`. In both cases, you see a test summary that displays the name of each test case and method that was run. You also see all the assertion failures that occurred. These include pointers to the line in the test code where the failure occurred. Assertion failures also list the expected value and actual value.

The test results have a format that is specific to the IDE that you are using. The test results format for Eclipse with ADT is

described in Testing from Eclipse with ADT (/tools/testing/testing_eclipse.html#RunTestEclipse). The test results format for tests run from the command line is described in Testing from Other IDEs (/tools/testing/testing_otheride.html#RunTestsCommand).

monkey and monkeyrunner

The SDK provides two tools for functional-level application testing:

- The UI/Application Exerciser Monkey, usually called "monkey", is a command-line tool that sends pseudo-random streams of keystrokes, touches, and gestures to a device. You run it with the Android Debug Bridge (adb) tool. You use it to stress-test your application and report back errors that are encountered. You can repeat a stream of events by running the tool each time with the same random number seed.
- The monkeyrunner tool is an API and execution environment for test programs written in Python. The API includes functions for connecting to a device, installing and uninstalling packages, taking screenshots, comparing two images, and running a test package against an application. Using the API, you can write a wide range of large, powerful, and complex tests. You run programs that use the API with the `monkeyrunner` command-line tool.

Working With Package names

In the test environment, you work with both Android application package names and Java package identifiers. Both use the same naming format, but they represent substantially different entities. You need to know the difference to set up your tests correctly.

An Android package name is a unique system name for a `.apk` file, set by the "android:package" attribute of the <manifest> element in the package's manifest. The Android package name of your test package must be different from the Android package name of the application under test. By default, Android tools create the test package name by appending ".test" to the package name of the application under test.

The test package also uses an Android package name to target the application package it tests. This is set in the "android:targetPackage" attribute of the <instrumentation> element in the test package's manifest.

A Java package identifier applies to a source file. This package name reflects the directory path of the source file. It also affects the visibility of classes and members to each other.

Android tools that create test projects set up an Android test package name for you. From your input, the tools set up the test package name and the target package name for the application under test. For these tools to work, the application project must already exist.

By default, these tools set the Java package identifier for the test class to be the same as the Android package identifier. You may want to change this if you want to expose members in the application under test by giving them package visibility. If you do this, change only the Java package identifier, not the Android package names, and change only the test case source files. Do not change the Java package name of the generated `R.java` class in your test package, because it will then conflict with the `R.java` class in the application under test. Do not change the Android package name of your test package to be the same as the application it tests, because then their names will no longer be unique in the system.

What to Test

The topic What To Test (/tools/testing/what_to_test.html) describes the key functionality you should test in an Android application, and the key situations that might affect that functionality.

Most unit testing is specific to the Android component you are testing. The topics Activity Testing (/tools/testing/activity_testing.html), Content Provider Testing (/tools/testing/contentprovider_testing.html), and Service Testing (/tools/testing/service_testing.html) each have a section entitled "What To Test" that lists possible testing areas.

When possible, you should run these tests on an actual device. If this is not possible, you can use the Android Emulator (/tools/devices/emulator.html) with Android Virtual Devices configured for the hardware, screens, and versions you want to test.

Next Steps

To learn how to set up and run tests in Eclipse, please refer to Testing from Eclipse with ADT (/tools/testing/testing_eclipse.html). If you're not working in Eclipse, refer to Testing from Other IDEs (/tools/testing/testing_otheride.html).

If you want a step-by-step introduction to Android testing, try the Activity Testing Tutorial (/tools/testing/activity_test.html).