Android Testing Fundamentals

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.2	4-Apr-2012	Added basic information on monkey, monkeyrunner, and Robotium.	KJ
0.1	18-Jan-2011	Initial version	KJ

Contents

1 Android Testing Guidelines **1**

1.1 Activity Testing . . . 1
1.2 Activity Testing (cont.) . . . 1
1.3 Activity Testing (cont.) . . . 2
1.4 Service Testing . . . 2
1.5 Service Testing (cont.) . . . 2
1.6 Content Provider Testing . . . 3
1.7 Other Testing: External Resource Dependencies . . . 3

2 Android Testing Environment **4**

2.1 Android Test Framework . . . 4
2.2 Android Test Project . . . 5
2.3 Android Test Case Classes . . . 5
2.4 Additional General-Purpose Asserts . . . 6
2.5 Additional View-Related Asserts . . . 6
2.6 Mock Object Classes . . . 6
2.7 Creating an Android Test Project in Eclipse . . . 8
2.8 The Test Project Manifest File . . . 8
2.9 Activity Testing: Activity Test Classes . . . 9
2.10 Overriding the Base Activity Test Class . . . 10
2.11 Managing Activity Lifecycle and the `Instrument` Class . . . 10
2.12 Turning Off Touch Mode . . . 10
2.13 Generating Touch Events . . . 11
2.14 Sending Key Events . . . 11
2.15 Testing on the UI Thread . . . 12
2.16 Running Tests from Eclipse . . . 13
2.17 Running Tests from the Command Line . . . 13
2.18 Service Testing . . . 14
2.19 Content Provider Testing . . . 15
2.20 Application Class Testing . . . 15

3 Stress Testing with the UI/Application Exerciser Monkey **16**

3.1 What is the Monkey? . . . 16

3.2 Basic Use of the Monkey . . . 16

3.3 Controlling Monkey Event Generation . . . 17

3.4 Controlling Monkey Debug Behavior . . . 17

4 Other Android Testing Tools **18**

4.1 The `monkeyrunner` Tool . . . 18

4.2 An Example monkeyrunner Script . . . 18

4.3 Robotium . . . 19

5 Resource List **20**

5.1 Books . . . 20

5.2 Web Sites . . . 20

Chapter 1

Android Testing Guidelines

What should I test?

1.1 Activity Testing

Screen Sizes and Resolutions Verify that interfaces display correctly on all target devices you support.

- Test it on all of the screen sizes and densities on which you want it to run.
- You can test the application on multiple sizes and densities using AVDs, or you can test your application directly on the devices that you are targeting.

Run-Time Configuration Changes Test that each activity responds correctly to the possible changes in the device's configuration while your application is running.

- These include a change to the device's:
  - Orientation
  - Language and region
  - Day/night mode
  - Docking status
  - Physical keyboard hidden or visible

1.2 Activity Testing (cont.)

Input Validation Test that each Activity responds correctly to input values.

- Verify key events are reflected correctly in the Views.
- Verify that context-sensitive Views change their state correctly in response to input.
- Verify error messages and dialogs in response to invalid input.
- Test different input methods, such as touch events.

1.3 Activity Testing (cont.)

Lifecycle Events Test that each Activity handles lifecycle events correctly.

- In general, lifecycle events are actions, either from the system or from the user, that trigger a callback method such as `onCreate()` or `onClick()`.
- For example, an Activity should respond to pause or destroy events by saving its state.
- Remember that even a change in screen orientation causes the current Activity to be destroyed, so you should test that accidental device movements don't accidentally lose the application state.

INTENTS

- Test that each Activity correctly handles the Intents listed in the intent filter specified in its manifest.
- Test valid and invalid permission, if you restrict access by permission.
- Test URI and extra data information associated with Intents. Verify correct behavior if URI or extras are missing or invalid.

1.4 Service Testing

Lifecycle Events Test that each Service handles lifecycle events correctly.

- Ensure that the `onCreate()` is called in response to `Context.startService()` or `Context.bindService()`.
- Ensure that `onDestroy()` is called in response to `Context.stopService()`, `Context.unbindService()`, `stopSelf(` or `stopSelfResult()`.
- Test that the Service correctly handles multiple calls from `Context.startService()`. Only the first call triggers `Service.onCreate()`, but all calls trigger a call to `Service.onStartCommand()`.
- Remember that `startService()` calls don't nest, so a single call to `Context.stopService()` or `Service.stopSelf()` (but not `stopSelf(int)`) will stop the Service. Test that the Service stops at the correct point.

1.5 Service Testing (cont.)

Business Logic Test any business logic that your Service implements. Business logic includes checking for invalid values, financial and arithmetic calculations, and so forth.

INTENTS

- Test that each Service correctly handles the Intents listed in the intent filter specified in its manifest.
- Test valid and invalid permission, if you restrict access by permission.
- Test URI and extra data information associated with Intents. Verify correct behavior if URI or extras are missing or invalid.

1.6 Content Provider Testing

Public Contract If you intend your provider to be public and available to other applications, you should test it as a contract.

- Test with constants that your provider publicly exposes. For example, look for constants that refer to column names in one of the provider's data tables. These should always be constants publicly defined by the provider.
- Test all the URIs offered by your provider. Your provider may offer several URIs, each one referring to a different aspect of the data.
- Test invalid URIs: Your unit tests should deliberately call the provider with an invalid URI, and look for errors. Good provider design is to throw an `IllegalArgumentException` for invalid URIs.

Standard Provider Interactions Test the standard provider interactions.

- Most providers offer six access methods: query(), insert(), delete(), update(), getType(), and onCreate().

Business Logic Test the business logic that your provider should enforce.

- Business logic includes handling of invalid values, financial or arithmetic calculations, elimination or combining of duplicates, and so forth.

1.7 Other Testing: External Resource Dependencies

If your application depends on network access, SMS, Bluetooth, or GPS, then you should test what happens when the resource or resources are not available.

You can use the emulator to test network access, bandwidth, and GSP. The test can be performed manually, or scripted using the emulator console.

Chapter 2

Android Testing Environment

2.1 Android Test Framework

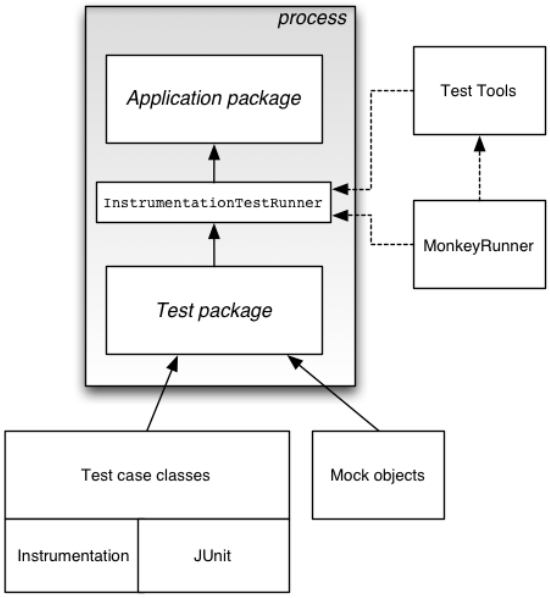

Figure 2.1: New Android Test Project Wizard

Android Test Projects Android test projects are test suites based on JUnit

- Android provides a set of JUnit extensions designed to test Android components such as Activities, Services, and Content Providers
- You can use plain JUnit to test an auxiliary class that doesn't call the Android API

Note
The Android testing API supports only JUnit 3 capabilities, not JUnit 4. For example, test methods must be named starting with `test` rather than annotated with `@Test`.

Other Android Test Tools Other tools provided by the SDK that you can use for for testing include:

- UI/Application Exerciser Monkey, a command-line tool for stress-testing UIs by sending pseudo-random events to a device
- The monkeyrunner tool, a Python API for writing programs that control an Android device or emulator from outside of Android code
- The AVD emulator console

2.2 Android Test Project

Android tests, like Android applications, are organized into projects.

- An Android test project is a directory or Eclipse project in which you create the source code, manifest file, and other files for a test package.
- An Android test project creates a test application (an `.apk` file) that you must install on the test device/emulator along with the target application to test.
- The Android test application contains special code enabling it to *instrument* the target application on the test device/emulator, so that it can be run under the control of the test application.

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 Android 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.

2.3 Android Test Case Classes

All Android test classes are in the `android.test` package. Some auxiliary classes are in sub-packages.

- `AndroidTestCase` is the base Android test class, derived from `junit.framework.TestCase`.
- In general, avoid using `AndroidTestCase` directly, and instead use one of its subclasses designed to test specific application components:

  Activity testing
  `ActivityInstrumentationTestCase2`
  `ActivityUnitTestCase`
  `SingleLaunchActivityTestCase`

  Service testing
  `ServiceTestCase`

  Content Provider testing
  `ProviderTestCase2`

  Application class testing
  `ApplicationTestCase`

2.4 Additional General-Purpose Asserts

The Android SDK includes the `MoreAsserts` class, which defines a variety of `assert` methods beyond those provided by JUnit 3, including:

- `assertEquals(Object[] expected, Object[] actual)`
- `assertEquals(int[] expected, int[] actual)`
- `assertEquals(double[] expected, double[] actual)`
- `assertEquals(Set<? extends Object> expected, Set<? extends Object> actual)`
- `assertMatchesRegex(String expectedRegex, String actual)`
- `assertNotMatchesRegex(String expectedRegex, String actual)`

Android also provides overloaded versions of these methods that take an additional `String message` as their first parameter to provide more meaningful reporting in case of failures.

2.5 Additional View-Related Asserts

To support Activity testing, the `ViewAsserts` class provides several `asserts` for testing Views, including:

- `assertGroupContains(ViewGroup parent, View child)`
- `assertGroupNotContains(ViewGroup parent, View child)`
- `assertHasScreenCoordinates(View origin, View view, int x, int y)`
- `assertOnScreen(View origin, View view)`
- `assertOffScreenAbove(View origin, View view)`
- `assertOffScreenBelow(View origin, View view)`

Android also provides overloaded versions of these methods that take an additional `String message` as their first parameter to provide more meaningful reporting in case of failures.

An example of using `assertOnScreen()`:

```
ViewAsserts.assertOnScreen(view1.getRootView(), view2);
```

2.6 Mock Object Classes

You can completely or partially isolate the application component under test through the use of *mock objects*.

- The Android SDK provides mock object classes for many system resource classes.
- By default, most or all of the methods implemented by these classes simply throw an `UnsupportedOperationException` if called.
- You can create a subclass of a mock object class and override just those methods required by the test target. For example, these methods could return hard-coded values, simple calculated values, or access information from alternative databases and/or locations on the file system.

The mock object classes are contained in the `android.test.mock` package. The classes provided are:

- `MockApplication`
- `MockContentProvider`
- `MockContentResolver`
- `MockContext`
- `MockCursor`
- `MockDialogInterface`
- `MockPackageManager`
- `MockResources`

Android provides two "pre-built" mock Context classes in the `android.test` package 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.

2.7 Creating an Android Test Project in Eclipse

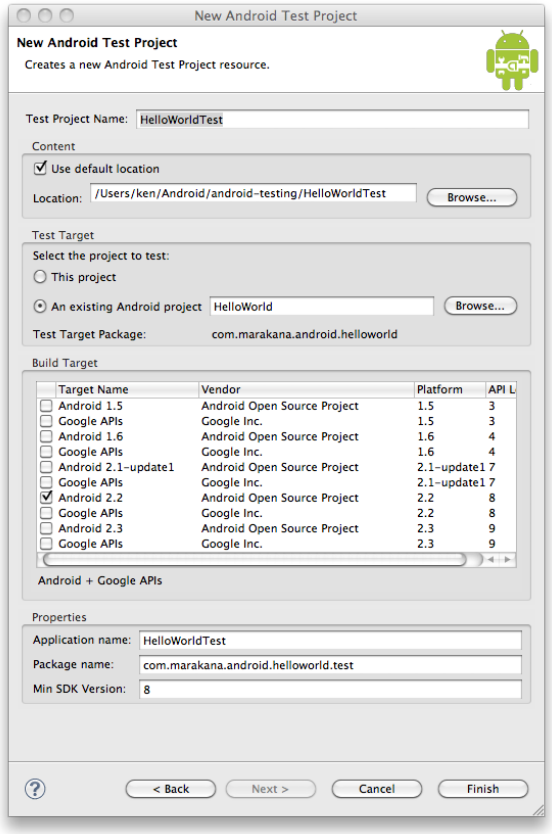

Figure 2.2: New Android Test Project Wizard

To create a test project in Eclipse:

1. In Eclipse, select File ⇒ New ⇒ Other to open the Select a Wizard dialog.
2. In the dialog, expand the entry for Android, then select Android Test Project and click `Next` to display the New Android Test Project wizard.
3. Next to `Test Project Name`, enter a name for the project. A typical convention is to append "Test" to the project name for the application under test.
4. In the Content panel, examine the suggested `Location` to the project and change if desired.
5. In the Test Target panel, set `An Existing Android Project`, click `Browse`, then select your Android application from the list. You now see that the wizard has completed the `Test Target Package`, `Application Name`, and `Package Name` fields for you.
6. In the Build Target panel, select the Android SDK platform that the application under test uses.
7. Click `Finish` to complete the wizard.

2.8 The Test Project Manifest File

As the Android test project creates an application that is installed on the test device/emulator, it must have a manifest file.

- The `<manifest>` element's `package` attribute should be a unique Android package name. By convention, it is the package name of the application under test with "`.test`" appended. This is the default provided by the Android tools when you create a test project.
- The test application has no Activities, Services, Content Providers, or Broadcast Receivers, so there should be no corresponding elements present in the manifest.
- The `<application>` element should contain the following element:

```
<uses-library android:name="android.test.runner" />
```

- The `<manifest>` element must also have an `<instrumentation>` child element, with the following attributes set:

`android:targetPackage` The Android package name of the application to test

`android:name` The JUnit test runner class, typically `android.test.InstrumentationTestRunner`

Here is an example of a complete manifest file for an Android test project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.marakana.android.currencyconverter.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="4" />
    <instrumentation android:targetPackage="com.marakana.android.currencyconverter"
        android:name="android.test.InstrumentationTestRunner" />
</manifest>
```

2.9 Activity Testing: Activity Test Classes

In practice, most Activity test cases are derived from:

`ActivityInstrumentationTestCase2`
Designed to do functional testing of one or more Activities in an application, using a normal system infrastructure

`ActivityUnitTestCase`
Designed to test a single Activity in isolation. Before you start the Activity, you can inject a mock Context or Application, or both.

`SingleLaunchActivityTestCase`
Designed for testing an activity that runs in a launch mode other than `standard`. It invokes `setUp()` and `tearDown()` only once, instead of once per test method call. It does not allow you to inject any mock objects.

ACTIVITY TEST CLASS HIERARCHY

- `ActivityInstrumentationTestCase2` and `ActivityUnitTestCase` are subclasses of `ActivityTestCase`.
- `ActivityTestCase` and `SingleLaunchActivityTestCase` are subclasses of `InstrumentationTestCase`, which in turn is a subclass of `junit.framework.TestCase`.

For more information on Activity launch modes, see the Android Manifest File reference documentation for the `<activity>` element's `android:launchMode` attribute.

2.10 Overriding the Base Activity Test Class

When creating an Activity test case derived from any of the Activity Test Classes, you must provide a no-argument constructor.

- The constructor must explicitly invoke the base class's constructor, providing the Activity class under test as an argument.
- For example:

```
public class HelloWorldActivityTest
        extends ActivityInstrumentationTestCase2<HelloWorldActivity> {

    public HelloWorldActivityTest() {
        super(HelloWorldActivity.class);
    }
}
```

2.11 Managing Activity Lifecycle and the `Instrument` Class

Android instrumentation is a set of control methods or "hooks" in the Android system.

- These hooks control an Android component independent of its normal lifecycle.
- The Activity test classes use instrumentation to manage an Activity under test, allowing you to invoke the callback methods of an Activity in your test code.

The `getActivity()` method returns the Activity under test, starting it if necessary.

- For each test method invocation, the Activity is not created until the first time this method is called.
- If you want to provide custom setup values to your Activity, you may do so before your first call to `getActivity()`.

The `InstrumentationTestCase` class provides a `getInstrumentation()` method, which returns an `Instrumentation` object, which provides other methods from controlling an Activity, such as invoking its lifecycle methods. For example:

- `callActivityOnPause(Activity activity)`
- `callActivityOnStop(Activity activity)`
- `callActivityOnRestart(Activity activity)`
- `callActivityOnStart(Activity activity)`

2.12 Turning Off Touch Mode

To control the emulator or a device with key events you send from your tests, you must turn off touch mode.

- If you do not do this, the key events are ignored.

To turn off touch mode:

- Invoke `ActivityInstrumentationTestCase2.setActivityInitialTouchMode(false)` before you call `getActi` to start the activity.

Note
You must invoke the method in a test method that is not running on the UI thread. For this reason, you can't invoke the touch mode method from a test method that is annotated with `@UIThread`. Instead, invoke the touch mode method from `setUp()`.

- For example:

```
public void setUp() {
    super.setUp();
    setActivityInitialTouchMode(false);
    this.activity = getActivity();
}
```

Thread management in test cases is discussed later in this module.

2.13 Generating Touch Events

The `TouchUtils` class provides many static methods for simulating touch events in the Activity. These include:

- `clickView(InstrumentationTestCase test, View v)`
- `drag(InstrumentationTestCase test, float fromX, float toX, float fromY, float toY, int stepCount)`
- `dragQuarterScreenDown(InstrumentationTestCase test, Activity activity)`
- `dragQuarterScreenUp(InstrumentationTestCase test, Activity activity)`
- `longClickView(InstrumentationTestCase test, View v)`
- `scrollToBottom(InstrumentationTestCase test, Activity activity, ViewGroup v)`
- `scrollToTop(InstrumentationTestCase test, Activity activity, ViewGroup v)`
- `tapView(InstrumentationTestCase test, View v)`

2.14 Sending Key Events

The `InstrumentTestCase` class provides methods for sending key events to the target Activity:

`public void sendKeys (String keysSequence)`
Sends a series of key events through instrumentation and waits for idle. The sequence of keys is a string containing the key names as specified in `KeyEvent`, without the `KEYCODE_` prefix. For instance:

```
sendKeys("DPAD_LEFT A B C DPAD_CENTER");
```

Each key can be repeated by using the `N*` prefix. For instance, to send two `KEYCODE_DPAD_LEFT`, use the following:

```
sendKeys("2*DPAD_LEFT");
```

`public void sendKeys (int... keys)`
Sends a series of key events through instrumentation and waits for idle. For instance:

```
sendKeys(KEYCODE_DPAD_LEFT, KEYCODE_DPAD_CENTER);
```

`public void sendRepeatedKeys (int... keys)`
Sends a series of key events through instrumentation and waits for idle. Each key code must be preceded by the number of times the key code must be sent. For instance:

```
sendRepeatedKeys(1, KEYCODE_DPAD_CENTER, 2, KEYCODE_DPAD_LEFT);
```

2.15 Testing on the UI Thread

An application's Activities run on the application's *UI thread* (also known as the *looper thread*).

The test application runs in a *separate* thread in the *same process* as the application under test.

- Generally, your test application can *read* properties and values from objects in the UI thread (for example, `TextView.getText()`).
- If your test application attempt to *change* properties on objects or otherwise send events to the UI thread from the test thread (for example, `TextView.setTest()`), you'll encounter a `WrongThreadException`.

The `TouchUtils` methods and the `InstrumentTestCase.sendKeys` methods have been written to send these events to the UI thread safely.

To otherwise manipulate the UI, you must explicitly execute that code in the UI thread. There are two ways to do so:

- To run an entire test method on the UI thread, annotate the thread with `@UIThreadTest`. Methods that do not interact with the UI are not allowed; for example, you can't invoke `Instrumentation.waitForIdleSync()`.
- To run a subset of a test method on the UI thread, create an anonymous class of type `Runnable`, put the statements you want in the `run()` method, and instantiate a new instance of the class as a parameter to the method *`activity`*`.runOnUiThread()`, where *`activity`* is the instance of the Activity you are testing. Execute `Instrumentation.waitForIdleSync()` afterwards, to wait until the Runnable has completed executed before continuing with your test method. For example:

```
activity.runOnUiThread(new Runnable() {
    public void run() {
        spinner.requestFocus();
    }
});

getInstrumentation().waitForIdleSync();

sendKeys(KeyEvent.KEYCODE_DPAD_CENTER);
```

2.16 Running Tests from Eclipse

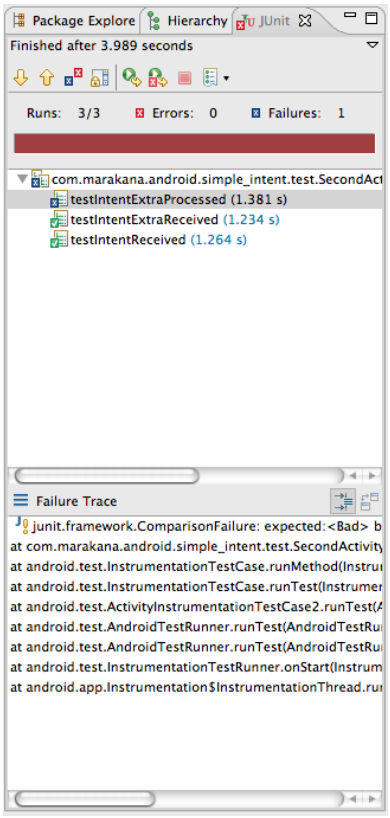

Figure 2.3: Android Test Results in Eclipse

In Eclipse with ADT, you run a test application as an "Android JUnit test" rather than a regular Android application.

To run the test application as an Android JUnit test, in the Package Explorer right-click the test project and select Run As ⇒ Android JUnit Test.

The ADT plugin then launches the test application and the application under test on a the target emulator or device. When both applications are running, the testing framework runs the tests and reports the results in the JUnit view of Eclipse, which appears by default as a tab next to the Package Explorer.

2.17 Running Tests from the Command Line

- To run a test from the command line, you run `adb shell` to start a command-line shell on your device or emulator, and then in the shell run the `am instrument` command.
- As a shortcut, you can start an `adb shell`, call `am instrument`, and specify command-line flags all on one input line. The shell opens on the device or emulator, runs your tests, produces output, and then returns to the command line on your computer.
- To run a test with `am instrument`:

  1. If necessary, rebuild your main application and test package.
  2. Install your test package and main application Android package files (`.apk` files) to your current Android device or emulator

3. At the command line, enter:

```
adb shell am instrument -w test_package_name/runner_class
```

test_package_name
The Android package name of your test application

runner_class
The name of the Android test runner class you are using, usually `android.test.InstrumentationTestRunner`

The `am instrument` tool passes testing options to `InstrumentationTestRunner` or a subclass in the form of key-value pairs, using the `-e` flag, with this syntax:

```
-e <key> <value>
```

Some keys accept multiple values, specified as a comma-separated list. Common keys include:

Key	Value	Description
`package`	*`Java_package_name`*	The fully-qualified Java package name for one of the packages in the test application. Any test case class that uses this package name is executed. Notice that this is not an Android package name; a test package has a single Android package name but may have several Java packages within it.
`class`	*`class_name`*	The fully-qualified Java class name for one of the test case classes. Only this test case class is executed.
	`class_name#method name`	A fully-qualified test case class name, and one of its methods. Only this method is executed. Note the hash mark (`#`) between the class name and the method name.
`size`	`small|medium|large`	Runs a test method annotated by size. The annotations are `@SmallTest`, `@MediumTest`, and `@LargeTest`.

2.18 Service Testing

`ServiceTestCase` extends the JUnit `TestCase` class with with methods for testing application permissions and for controlling the application and Service under test.

- It also provides mock application and Context objects that isolate your test from the rest of the system.

`ServiceTestCase` defers initialization of the test environment until you call `ServiceTestCase.startService()` or `ServiceTestCase.bindService()`.

- This allows you to set up your test environment, particularly your mock objects, before the Service is started.

The `setUp()` method for `ServiceTestCase` is called before each test.

- It sets up the test fixture by making a copy of the current system Context before any test methods touch it.
- You can retrieve this Context by calling `getSystemContext()`.
- If you override this method, you must call `super.setUp()` as the first statement in the override.

The methods `setApplication()` and `setContext(Context)` allow you to set a mock Context or mock Application (or both) for the Service, before you start it.

2.19 Content Provider Testing

You test a Content Provider with a subclass of `ProviderTestCase2`.

- This base class extends `AndroidTestCase`.

The `ProviderTestCase2` constructor performs several important initialization steps for creating an isolated test environment.

- All subclasses must invoke the super constructor.

The `ProviderTestCase2` constructor creates an `IsolatedContext` object that allows file and database operations but stubs out other interactions with the Android system.

- The file and database operations themselves take place in a directory that is local to the device or emulator and has a special prefix.

The constructor then creates a `MockContentResolver` to use as the resolver for the test.

Lastly, the constructor creates an instance of the provider under test.

- This is a normal `ContentProvider` object, but it takes all of its environment information from the `IsolatedContext`, so it is restricted to working in the isolated test environment.
- All of the tests done in the test case class run against this isolated object.

2.20 Application Class Testing

You use the `ApplicationTestCase` test case class to test the setup and teardown of Application objects.

- Application 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
This test case does not allow you to control testing of the components within your application package.

Chapter 3

Stress Testing with the UI/Application Exerciser Monkey

3.1 What is the Monkey?

The Monkey is a command-line tool that you can run on an emulator or physical device to perform a *stress test*.

- It generates a pseudo-random stream of user events.
- You can configure the number and type of events, as well as other debugging options.

While executing, the Monkey monitors for three special conditions, which it can then handle:

- If you have constrained the Monkey to run in one or more specific packages, it blocks attempts to navigate to any other packages.
- If your application crashes or receives an unhandled exception, the Monkey stops and reports the error.
- If your application generates an *application not responding error*, the Monkey stops and reports the error.

Tip
See the UI/Application Exerciser Monkey section of the Android Developer's Guide for more information on the Monkey.

3.2 Basic Use of the Monkey

To launch the Monkey use `adb shell` to invoke the `monkey` command on the device or emulator:

```
adb shell monkey [options] <event-count>
```

With no options specified, the Monkey sends events to any (and all) packages installed on your target.

- More typically, you'll use the `-p` option to restrict the Monkey to one or more application packages.
- You can also use the `-v` option to generate verbose shell output of the Monkey's actions.
- For example:

```
adb shell monkey -v -p com.marakana.android.app1 -p com.marakana.android.app2
```

3.3 Controlling Monkey Event Generation

The `monkey` command supports options for adjusting the percentage of different event types generated by the Monkey:

`--pct-touch <percent>`
A down-up event in a single place on the screen

`--pct-motion <percent>`
A down event somewhere on the screen, a series of pseudo-random movements, and an up event

`--pct-trackball <percent>`
One or more random trackball movements, sometimes followed by a click

`--pct-nav <percent>`
Up/down/left/right, as input from a directional input device

`--pct-majornav <percent>`
Navigation events that typically cause actions within your UI, such as the center button in a 5-way pad, the back key, or the menu key

`--pct-syskeys <percent>`
Keys such as Home, Back, Start Call, End Call, or Volume controls

`--pct-appswitch <percent>`
Activity launches via a `startActivity()` call

`--pct-anyevent <percent>`
Other types of events such as keypresses and other less-used buttons on the device

3.4 Controlling Monkey Debug Behavior

The `money` command supports other options to change its default behavior in response to exceptions, timeouts, and crashes, including:

`--ignore-crashes`
Ignore application crashes and unhandled exceptions, rather than terminating event generation

`--ignore-timeouts`
Ignore application not responding conditions, rather than terminating event generation

`--ignore-security-exceptions`
Ignore permission errors, rather than terminating event generation

`--kill-process-after-error`
Kill the process after an error, rather than leaving it running after the test

`--wait-dbg`
Stop the Monkey from executing until a debugger is attaches to the process

Chapter 4

Other Android Testing Tools

4.1 The `monkeyrunner` Tool

The `monkeyrunner` tool provides a Python API for writing programs that control an Android device or emulator to automate actions such as:

- Installing an Android application or test package
- Running an application
- Sending keystrokes to it
- Taking screenshots of its user interface
- Storing screenshots on the workstation

The `monkeyrunner` tool supports:

- Multiple device control, allowing you to run tests concurrently on multiple devices
- Functional testing, automating the start-to-finish test of an entire Android application
- Regression testing, with the ability to take screen shots and compare against reference versions of those screens
- Extensible automation, through the integration of additional standard and custom Python modules

See the `monkeyrunner` section of the Android Developer's Guide for more information on the `monkeyrunner`.

Note
The `monkeyrunner` tool is not related to the UI/Application Exerciser Monkey (also known as the Monkey tool), which is used for stress testing.

4.2 An Example monkeyrunner Script

The following script demonstrates several monkeyrunner capabilities:

```
# Imports the monkeyrunner modules used by this program
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice

# Connects to the current device, returning a MonkeyDevice object
device = MonkeyRunner.waitForConnection()

# Installs the Android package. Notice that this method returns a boolean, so you can test
# to see if the installation worked.
device.installPackage('myproject/bin/MyApplication.apk')

# sets a variable with the package's internal name
package = 'com.example.android.myapplication'

# sets a variable with the name of an Activity in the package
activity = 'com.example.android.myapplication.MainActivity'

# sets the name of the component to start
runComponent = package + '/' + activity

# Runs the component
device.startActivity(component=runComponent)

# Presses the Menu button
device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)

# Takes a screenshot
result = device.takeSnapshot()

# Writes the screenshot to a file
result.writeToFile('myproject/shot1.png','png')
```

4.3 Robotium

The Robotium test framework extends the basic JUnit-based Android test framework to perform more *black-box* and *feature* testing. Features include the ability to develop and run tests:

- Without knowing specific view IDs
- Without access to the application source (test directly from an APK)
- That span multiple application activities

For more information on Robotium, visit the project's home page at http://robotium.org

Chapter 5

Resource List

5.1 Books

[1] Beck, Kent. JUnit Pocket Guide Sebastopol, Calif.: O'Reilly, 2004.

5.2 Web Sites

[2] Android Developers. http://d.android.com

[3] JUnit. http://www.junit.org

[4] Robotium. http://robotium.org