PART IV – Advanced Development

CHAPTER 15

Testing Your Patience

Presumably, you will want to test your code, beyond just playing around with it yourself by hand.

To that end, Android includes the JUnit test framework in the SDK, along with special test classes that will help you build test cases that exercise Android components, like activities and services. Even better, Android 1.5 has "gone the extra mile" and can pre-generate your test harness for you, to make it easier for you to add in your own tests.

This chapter assumes you have some familiarity with JUnit, though you certainly do not need to be an expert. You can learn more about JUnit at the JUnit site, from various books, and from the JUnit Yahoo forum.

You Get What They Give You

When you create a project in Android 1.5 using android create project, Android automatically creates a new `tests/` directory inside the project directory. If you look in there, you will see a complete set of Android project artifacts: manifest, source directories, resources, etc. This is actually a test project, designed to partner with the main project to create a complete testing solution.

In fact, that test project is all ready to go, other than not having any tests of significance. If you build and install your main project (onto an emulator or

device), then build and install the test project, you will be able to run unit tests.

Android ships with a very rudimentary JUnit runner, called `InstrumentationTestRunner`. Since this class resides in the Android environment (emulator or device), you need to invoke the runner to run your tests on the emulator or device itself. To do this, you can run the following command from a console:

```
adb shell am instrument -w
com.commonsware.android.database.tests/android.test.InstrumentationTestRunner
```

In this case, we are instructing Android to run all the available test cases for the `com.commonsware.android.database` package, as this chapter uses some tests implemented on the `Database/Contacts` sample project.

If you were to run this on your own project, substituting in your package name, with just the auto-generated test files, you should see results akin to:

```
com.commonsware.android.database.ContactsDemoTest:.
Test results for InstrumentationTestRunner=.
Time: 0.61

OK (1 test)
```

The first line will differ, based upon your package and the name of your project's initial activity, but the rest should be the same, showing that a single test was run, successfully.

Of course, this is only the beginning.

Erecting More Scaffolding

Here is the source code for the test case that Android automatically generates for you:

```
package com.commonsware.android.database;
```

```
import android.test.ActivityInstrumentationTestCase;

/**
 * This is a simple framework for a test of an Application. See
 * {@link android.test.ApplicationTestCase ApplicationTestCase} for more
information on
 * how to write and extend Application tests.
 * <p/>
 * To run this test, you can type:
 * adb shell am instrument -w \
 * -e class com.commonsware.android.database.ContactsDemoTest \
 * com.commonsware.android.database.tests/android.test.InstrumentationTestRunner
 */
public class ContactsDemoTest extends
ActivityInstrumentationTestCase<ContactsDemo> {

 public ContactsDemoTest() {
 super("com.commonsware.android.database", ContactsDemo.class);
 }

}
```

As you can see, there are no actual test methods. Instead, we have an `ActivityInstrumentationTestCase` implementation named `ContactsDemoTest`. The class name was generated by adding Test to the end of the main activity (`ContactsDemo`) of the project.

In the next section, we will examine `ActivityInstrumentationTestCase` more closely and see how you can use it to, as the name suggests, test your activities.

However, you are welcome to create ordinary JUnit test cases in Android – after all, this is just JUnit, merely augmented by Android. So, you can create classes like this:

```
package com.commonsware.android.database;

import junit.framework.TestCase;

public class SillyTest extends TestCase {
  protected void setUp() throws Exception {
    super.setUp();

    // do initialization here, run on every test method
  }

  protected void tearDown() throws Exception {
```

```
    // do termination here, run on every test method

    super.tearDown();
  }

  public void testNonsense() {
    assertTrue(1==1);
  }
}
```

There is nothing Android-specific in this test case. It is simply standard JUnit, albeit a bit silly.

You can also create test suites, to bundle up sets of tests for execution. Here, though, if you want, you can take advantage of a bit of Android magic: `TestSuiteBuilder`. `TestSuiteBuilder` uses reflection to find test cases that need to be run, as shown below:

```
package com.commonsware.android.database;

import android.test.suitebuilder.TestSuiteBuilder;
import junit.framework.Test;
import junit.framework.TestSuite;

public class FullSuite extends TestSuite {
  public static Test suite() {
    return(new TestSuiteBuilder(FullSuite.class)
                .includeAllPackagesUnderHere()
                .build());
  }
}
```

Here, we are telling Android to find all test cases located in `FullSuite`'s package (`com.commonsware.android.database`) and all sub-packages, and to build a `TestSuite` out of those contents.

A test suite may or may not be necessary for you. The command shown above to execute tests will execute any test cases it can find for the package specified on the command line. If you want to limit the scope of a test run, though, you can use the `-e` switch to specify a test case or suite to run:

```
adb shell am instrument -e class
com.commonsware.android.database.ContactsDemoTest -w
com.commonsware.android.database.tests/android.test.InstrumentationTestRunner
```

Here, we indicate we only want to run `ContactsDemoTest`, not all test cases found in the package.

Testing Real Stuff

While ordinary JUnit tests are certainly helpful, they are still fairly limited, since much of your application logic may be tied up in activities, services, and the like.

To that end, Android has a series of `TestCase` classes you can extend designed specifically to assist in testing these sorts of components.

ActivityInstrumentationTestCase

The test case created by Android's SDK tools, `ContactsDemoTest` in our example, is an `ActivityInstrumentationTestCase`. This class will run your activity for you, giving you access to the `Activity` object itself. You can then:

- Access your widgets
- Invoke public and package-private methods (more on this below)
- Simulate key events

Of course, the automatically-generated `ActivityInstrumentationTestCase` does none of that, since it does not know much about your activity. Below you will find an augmented version of `ContactsDemoTest` that does a little bit more:

```
package com.commonsware.android.database;

import android.test.ActivityInstrumentationTestCase;
import android.widget.ListView;
import android.widget.Spinner;

public class ContactsDemoTest
  extends ActivityInstrumentationTestCase<ContactsDemo> {
  private ListView list=null;
  private Spinner spinner=null;

  public ContactsDemoTest() {
```

```
    super("com.commonsware.android.database",
          ContactsDemo.class);
  }

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    ContactsDemo activity=getActivity();

    list=(ListView)activity.findViewById(android.R.id.list);
    spinner=(Spinner)activity.findViewById(R.id.spinner);
  }

  public void testSpinnerCount() {
    assertTrue(spinner.getAdapter().getCount()==3);
  }

  public void testListDefaultCount() {
    assertTrue(list.getAdapter().getCount()>0);
  }
}
```

Here are the steps to making use of `ActivityInstrumentationTestCase`:

1. Extend the class to create your own implementation. Since `ActivityInstrumentationTestCase` is a generic, you need to supply the name of the activity being tested (e.g., `ActivityInstrumentationTestCase<ContactsDemo>`).
2. In the constructor, when you chain to the superclass, supply the name of the package of the activity plus the activity class itself. You can optionally supply a third parameter, a boolean indicating if the activity should be launched in touch mode or not.
3. In `setUp()`, use `getActivity()` to get your hands on your `Activity` object, already typecast to the proper type (e.g., `ContactsDemo`) courtesy of our generic. You can also at this time access any widgets, since the activity is up and running by this point.
4. If needed, clean up stuff in `tearDown()`, no different than with any other JUnit test case/
5. Implement test methods to exercise your activity. In this case, we simply confirm that the `Spinner` has three items in its drop-down list and there is at least one contact loaded into the `ListView` by

default. You could, however, use `sendKeys()` and the like to simulate user input.

If you are looking at your emulator or device while this test is running, you will actually see the activity launched on-screen. `ActivityInstrumentationTestCase` creates a true running copy of the activity. This means you get access to everything you need; on the other hand, it does mean that the test case runs slowly, since the activity needs to be created and destroyed for each test method in the test case. If your activity does a lot on startup and/or shutdown, this may make running your tests a bit sluggish.

Note that your `ActivityInstrumentationTestCase` resides in the same package as the Activity it is testing – `ContactsDemoTest` and `ContactsDemo` are both in `com.commonsware.android.database`, for example. This allows `ContactsDemoTest` to access both public and package-private methods and data members. `ContactsDemoTest` still cannot access private methods, though. This allows `ActivityInstrumentationTestCase` to behave in a white-box (or at least gray-box) fashion, inspecting the insides of the tested activities in addition to testing the public API.

Now, despite the fact that Android's own tools create an `ActivityInstrumentationTestCase` subclass for you, that class is officially deprecated. They advise using `ActivityInstrumentationTestCase2` instead, which offers the same basic functionality, with a few extras, such as being able to specify the `Intent` that is used to launch the activity being tested. This is good for testing search providers, for example.

AndroidTestCase

For tests that only need access to your application resources, you can skip some of the overhead of `ActivityInstrumentationTestCase` and use `AndroidTestCase`. In `AndroidTestCase`, you are given a `Context` and not much more, so anything you can reach from a `Context` is testable, but individual activities or services are not.

While this may seem somewhat useless, bear in mind that a lot of the static testing of your activities will come in the form of testing the layout: are the widgets identified properly, are they positioned properly, does the focus work, etc. As it turns out, none of that actually needs an `Activity` object – so long as you can get the inflated `View` hierarchy, you can perform those sorts of tests.

For example, here is an `AndroidTestCase` implementation, `ContactsDemoBaseTest`:

```
package com.commonsware.android.database;

import android.test.AndroidTestCase;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ListView;
import android.widget.Spinner;

public class ContactsDemoBaseTest extends AndroidTestCase {
  private ListView list=null;
  private Spinner spinner=null;
  private ViewGroup root=null;

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    LayoutInflater inflater=LayoutInflater.from(getContext());

    root=(ViewGroup)inflater.inflate(R.layout.main, null);
    root.measure(480, 320);
    root.layout(0, 0, 480, 320);

    list=(ListView)root.findViewById(android.R.id.list);
    spinner=(Spinner)root.findViewById(R.id.spinner);
  }

  public void testExists() {
    assertNotNull(list);
    assertNotNull(spinner);
  }

  public void testRelativePosition() {
    assertTrue(list.getTop()>=spinner.getBottom());
    assertTrue(list.getLeft()==spinner.getLeft());
    assertTrue(list.getRight()==spinner.getRight());
  }
}
```

Most of the complicated work is performed in `setUp()`:

1. Inflate our layout using a `LayoutInflater` and the `Context` supplied by `getContext()`
2. Measure and lay out the widgets in the inflated `View` hierarchy – in this case, we lay them out on a 480x320 screen
3. Access the individual widgets to be tested

At that point, we can test static information on the widgets, but we cannot cause them to change very easily (e.g., we cannot simulate keypresses). In the case of `ContactsDemoBaseTest`, we simply confirm the widgets exist and are laid out as expected. We could use `FocusFinder` to test whether focus changes from one widget to the next should work as expected. We could ensure our resources exist under their desired names, test to see if our fonts exist in our assets, or anything else we can accomplish with just a `Context`.

Since we are not creating and destroying activities with each test case, these tests should run substantially faster.

Other Alternatives

Android also offers various other test case base classes designed to assist in testing Android components, such as:

- `ServiceTestCase`, used for testing services, as you might expect given the name
- `ActivityUnitTestCase`, a `TestCase` that creates the `Activity` (like `ActivityInstrumentationTestCase`), but does not fully connect it to the environment, so you can supply a mock `Context`, a mock `Application`, and other mock objects to test out various scenarios
- `ApplicationTestCase`, for testing custom `Application` subclasses

Monkeying Around

Independent from the JUnit system is the Monkey.

The Monkey is a test program that simulates random user input. It is designed for "bash testing", confirming that no matter what the user does, the application will not crash. The application may have odd results – random input entered into a Twitter client may, indeed, post that random input to Twitter. The Monkey does not test to make sure that results of random input make sense; it only tests to make sure random input does not blow up the program.

You can run the Monkey by setting up your initial starting point (e.g., the main activity in your application) on your device or emulator, then running a command like this:

```
adb shell monkey -p com.commonsware.android.database -v --throttle 100 600
```

Working from right to left, we are asking for 600 simulated events, throttled to run every 100 milliseconds. We want to see a list of the invoked events (`-v`) and we want to throw out any event that might cause the Monkey to leave our application, as determined by the application's package (`-p com.commonsware.android.database`).

The Monkey will simulate keypresses (both QWERTY and specialized hardware keys, like the volume controls), D-pad/trackball moves, and sliding the keyboard open or closed. Note that the latter may cause your emulator some confusion, as the emulator itself does not itself actually rotate, so you may end up with your screen appearing in landscape while the emulator is still, itself, portrait. Just rotate the emulator a couple of times (e.g., `<Ctrl>-<F12>`) to clear up the problem.

For playing with a Monkey, the above command works fine. However, if you want to regularly test your application this way, you may need some measure of repeatability. After all, the particular set of input events that trigger your crash may not come up all that often, and without that repeatable scenario, it will be difficult to repair the bug, let alone test that the repair worked.

To deal with this, the Monkey offers the `-s` switch, where you provide a seed for the random number generator. By default, the Monkey creates its own

seed, giving totally random results. If you supply the seed, while the sequence of events is random, it is random for that seed – repeatedly using the same seed will give you the same events. If you can arrange to detect a crash and know what seed was used to create that crash, you may well be able to reproduce the crash.

There are many more Monkey options, to control the mix of event types, to generate profiling reports as tests are run, and so on. The Monkey documentation in the SDK's Developer's Guide covers all of that and more.

Keyword Index

Class.....

Command.....

Constant.....

Method.....