Chapter 1

Introduction

For a while now, Agile development has been problematic for Android developers. There have been a number of ways to test the user interface (UI), such as Robotium or Monkey Runner, but before Android Studio 1.1, unit testing was hard to use, hard to configure, and quite challenging to implement on the Android platform.

Google would argue, no doubt, that in the past you could use JUnit3-style unit testing. But for anyone from classic Java development this was a dramatic backward step in time. Developers would stumble along hacking together a JUnit4 development environment using a number of third-party tools. More likely than not they would simply give up as the ever-increasing series of mutually incompatible library dependencies would finally wear them down. Because there simply wasn't the toolbox for the Android developer, Agile development on the mobile platform was immature and reminiscent of Java development in the early 2000s.

Thankfully all this has changed - Android now supports JUnit4 and Android developers can now return to unit testing. It's early days yet in the world of Android JUnit4 testing world and the documentation is thin on the ground, so in this book we're going to show practical ways to get your unit testing up and running using Android Studio. We'll also look at how this can be complemented by other UI-specific Android testing libraries such as Espresso to create a complete Agile testing framework for Android developers.

Hello, World Unit Test

Before we go any further let's look at a simple unit test. For demonstration purposes we can use the Add method from the Google Calculator example, which is available from https://github.com/googlesamples/android-testing (see Listing 1-1).

Listing 1-1. Add Method from Google's Calculator Example

```
public double add(double firstOperand, double secondOperand) {
    return firstOperand + secondOperand;
}
```

Listing 1-2 shows a very simple unit test, which tests if the Add method can add two numbers correctly.

Listing 1-2. Test Method for Add Method from Calculator Example

```
@Test
public void calculator_CorrectAdd_ReturnsTrue() {
    double resultAdd = mCalculator.add(3, 4);
    assertEquals(7, resultAdd, 0);
}
```

Unit tests use assertions to make sure the method provides an expected result. In this case we're using `assertEquals` to see if the `Add` method returns 7 when adding 3 to 4. If the test works, then we should see a positive or green result, and if it doesn't, then we'll see a red result in Android Studio.

Understand the Benefits of Using an Agile Approach to Android Development

If you're new to Agile development you're probably wondering how Agile can improve the development process.

At its most basic, Agile, and unit testing in particular, helps you to

- Catch more mistakes, earlier in the development process
- Confidently make more changes
- Build in regression testing
- Extend the life of your codebase

If you write unit tests and they cover a significant portion of your code then you're going to catch more bugs. You can make simple changes to tidy up the code or more extensive architectural changes, run your unit tests, and, if they all pass, be confident that you didn't introduce any subtle defects. The more unit tests you write, the more you can regression test your app whenever you change the code without fear. And once you have a lot of unit tests, then it becomes a regression test suite that allows you to have the confidence to do things you wouldn't otherwise attempt.

Unit tests mean you no longer have to program with a "leave well enough alone" mind-set. You can now make significant changes (changing to a new database, updating your back-end application programming interface (API), changing to a new material design theme, etc.) and be confident that your app is behaving the same as before you made the changes since all the tests execute without any errors.

Explore the Agile Testing Pyramid for Android

There are several types of tests you need in your test suite to make sure your app is fully tested. You should have Unit Tests for the component- or method-level functionality, API or Acceptance Tests for any back-end RESTful APIs, and GUI (graphical user interface) Tests for Android activities and general application workflow.

The classic Agile Test Pyramid first appeared in Succeeding with Agile *by Mike Cohn (*Pearson Education, 2010). This is a good guide for the relative quantity of each type of test your app is going to need (see Figure 1-1).

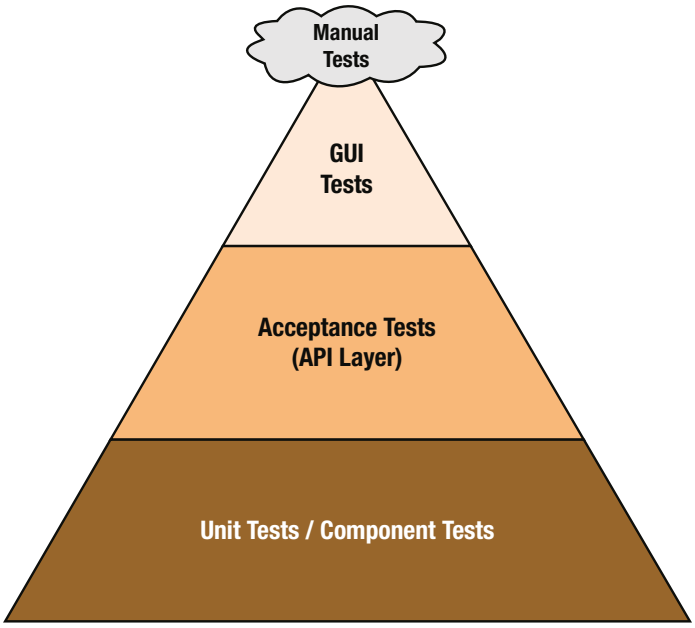

Figure 1-1. *Agile Test Pyramid*

Create Hello World Unit Test in Android

In the following example we show how to create our simple unit test example in Android Studio. This should return true assuming adding two numbers in the calculator Android app works correctly.

To set up and run a unit test you need to perform the following tasks:

- Prerequisites: Android Plugin for Gradle version 1.1.x
- Create the `src/test/java` folders
- Add JUnit:4:12 dependency in `build.gradle` (app) file
- Choose unit tests' test artifact in Build Variant
- Create unit tests
- Right-click tests to run tests

Click File ➤ Project Structure and make sure the Android Plugin version is greater than 1.1. In Figure 1-2 the Android Plugin version is 1.2.3 so we're good to go.

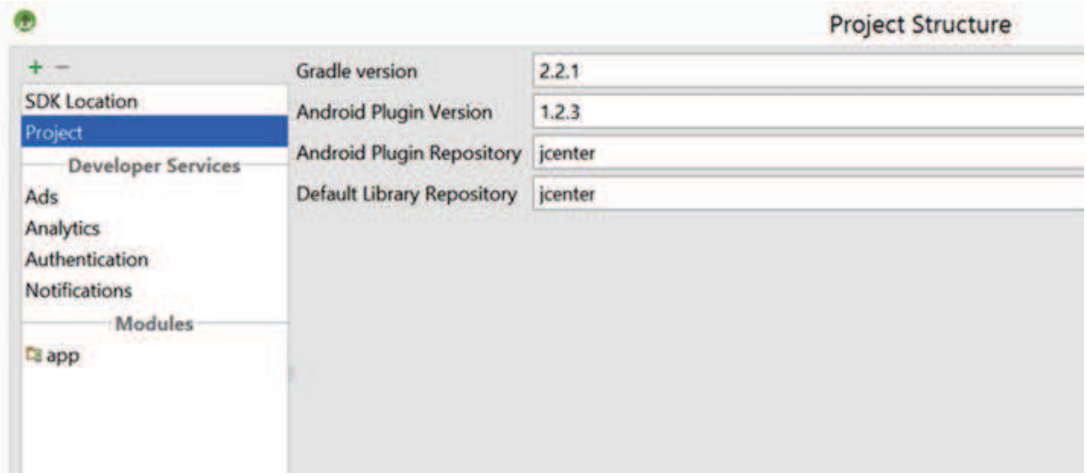

Figure 1-2.

Next we need to create the `src/test/java` folders for our unit test code. For the moment this seems to be hard-coded to this directory. So change to Project view to see the file structure and create the folders (see Figure 1-3). Alternatively, in Windows create the folders using the file explorer or on a Mac use the command line on a terminal window to make the changes. Don't be worried if the folders don't show up when you go back to the Android view in Android Studio. They'll show up when we change to unit tests in the Build Variant window.

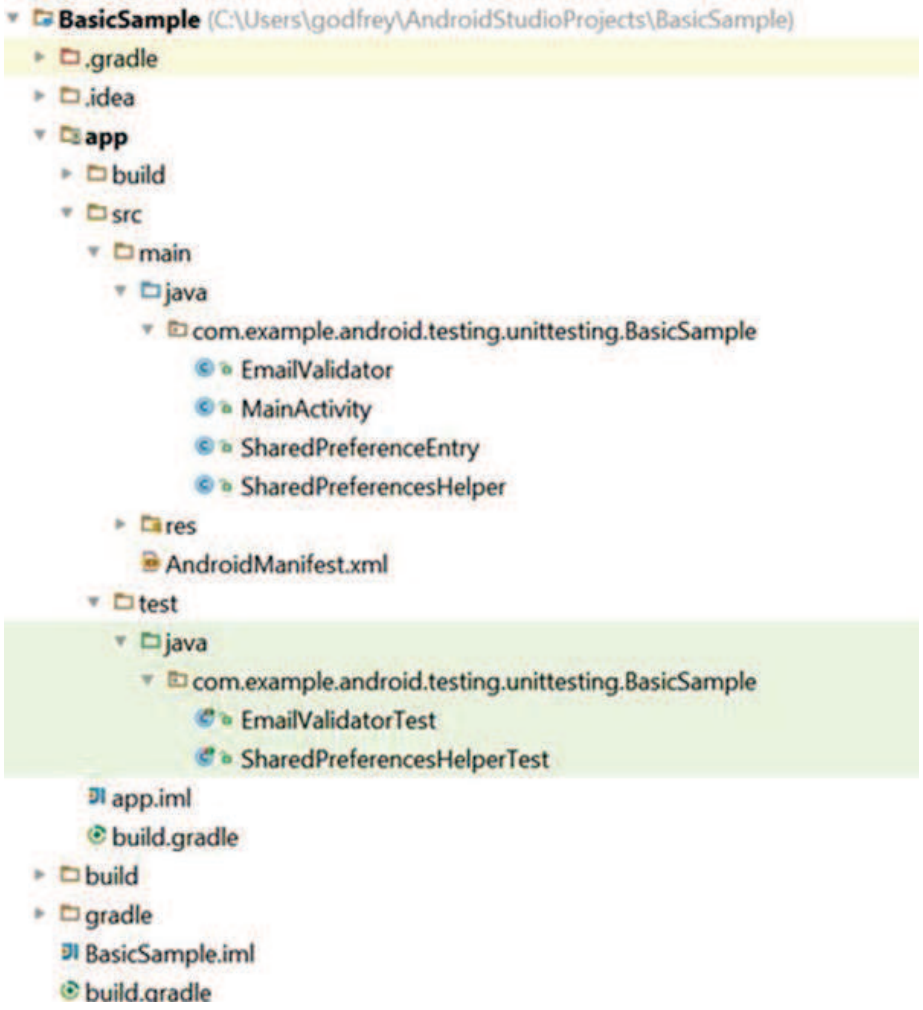

Figure 1-3. *Change to Project view*

Add `junit` library to the dependencies section in the `build.gradle` (app) file as shown in Figure 1-4.

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 22
    buildToolsVersion '22.0.1'
    defaultConfig {
        applicationId "com.example.android.testing.unittesting.BasicSample"
        minSdkVersion 8
        versionCode 1
        versionName "1.0"
        targetSdkVersion 22
    }
    productFlavors {
    }
}

dependencies {
    // Unit testing dependencies.
    testCompile 'junit:junit:4.12'
    testCompile 'org.mockito:mockito-core:1.10.19'
}
```

***Figure 1-4.** Modify the build.gradle file*

Choose the Unit Tests test artifact in Build Variants and use the debug build (see Figure 1-5). The test code directory should now also appear when you're in the Android view of your app.

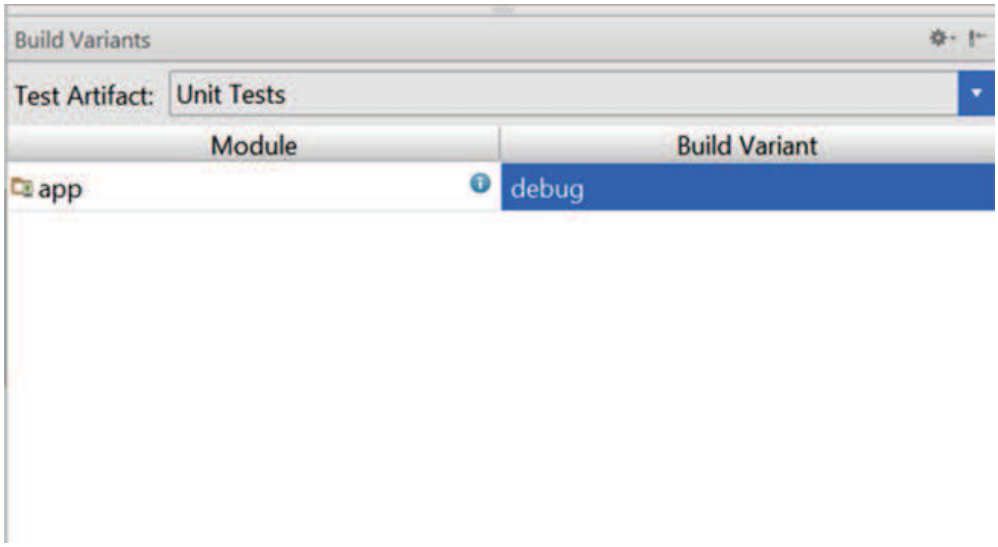

***Figure 1-5.** Choose Unit Tests in Build Variant*

Create the Unit Tests code for our simple example. We need to import the `org.junit.Before` so we can create a `Calculator` object. We need to import `org.junit.Test` to tell Android Studio that we're doing unit tests. And as we're going to do an `assertEquals`, we also need to import `org.junit.Assert.assertEquals` (see Listing 1-3).

Listing 1-3. Unit Test Code

```
package com.riis.calculatoradd;

import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

    private Calculator mCalculator;

    @Before
    public void setUp() {
        mCalculator = new Calculator();
    }

    @Test
    public void calculator_CorrectAdd_ReturnsTrue() {
        double resultAdd = mCalculator.add(3, 4);
        assertEquals("adding 3 + 4 didn't work this time", 7, resultAdd , 0);
            }
        }
```

Right-click the `CalculatorTest` java file and choose Run 'CalculatorTest' to run tests (see Figure 1-6).

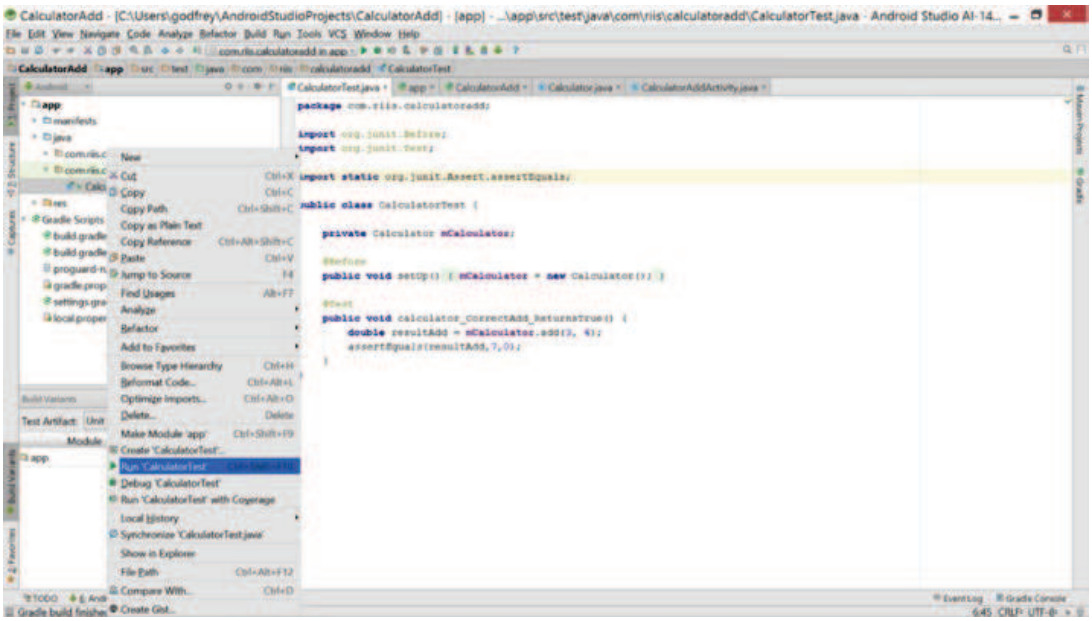

Figure 1-6. Running the unit test

You can see the results of the tests in the Run windows (see Figure 1-7). You may also want to click the configuration gear and choose Show Statistics to see how long the tests take.

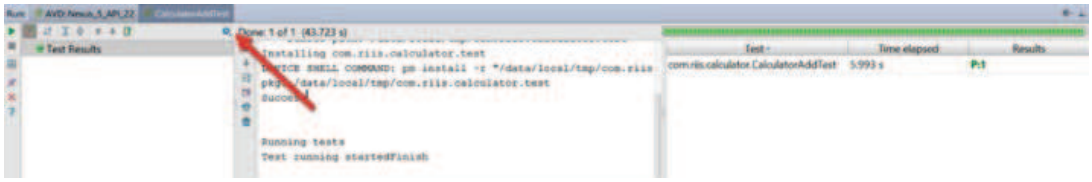

Figure 1-7. Test results

If your tests are successful they show as green, and anything that produces an error is shown in red. All your tests should be green before you continue with any coding.

GUI Tests

The real beauty of unit testing is that you don't need an emulator or physical device to do your testing. But, if we look back at our Agile Testing Pyramid (Figure 1-1) we know that we're going to need some GUI tests. Remember, GUI tests are tests on Activities and unit tests are tests on individual methods in your code. We won't need as many GUI tests as unit tests, but we're still going to have to test every activity for happy paths as well as not so happy paths.

When it comes to testing GUI we have a few frameworks that we can choose from: we can use the Android JUnit3 framework, Google's Espresso, UIAutomator, Robotium, or some Cucumber-type Android framework such as Calabash. In this book we'll use Google's Espresso as it's quick and easy to set up and it also has support for Gradle and Android Studio. But your author has used the other frameworks in the past and they all have their benefits.

Espresso has three components: ViewMatchers, ViewActions, and ViewAssertions. ViewMatchers are used to find a view, ViewActions allow you to do something with a view, and ViewAssertions are similar to unit test assertions—they let you assert that the value in the view is what you'd expect or not.

Listing 1-4 shows a simple example of an Espresso GUI test. We're adding two numbers again, but this time we're doing it by interacting with the GUI, not calling the underlying method.

Listing 1-4. Adding Two Numbers Using Espresso

```
public void testCalculatorAdd() {

     onView(withId(R.id.operand_one_edit_text)).perform(typeText(THREE));
     onView(withId(R.id.operand_two_edit_text)).perform(typeText(FOUR));
     onView(withId(R.id.operation_add_btn)).perform(click());
     onView(withId(R.id.operation_result_text_view)).check(matches(withText
     (RESULT)));
}
```

In this example `withId(R.id.operand_one_edit_text)` is one of the ViewMatchers in the code and `perform(typeText(THREE)` is a ViewAction. Finally `check(matches(withText(RESULT))` is the ViewAssertion.

Create Hello, World GUI Test

This time we show how to create our simple GUI test example in Android Studio. As with the unit test, this one should return true assuming that adding two numbers in the calculator Android app works correctly.

To set up and run a GUI test you need to perform the following tasks:

- Prerequisites: install the Android Support Repository
- Put the test classes in the `src/androidTest/java` folders
- Add Espresso dependency in `build.gradle` (app) file

- Choose Android Test Instrumentation Test Artifact in Build Variant
- Create GUI tests
- Right-click tests to run tests

Click Tools ➤ Android ➤ SDK Manager, click the SDK tools tab, and make sure the Android Support Repository is installed (see Figure 1-8).

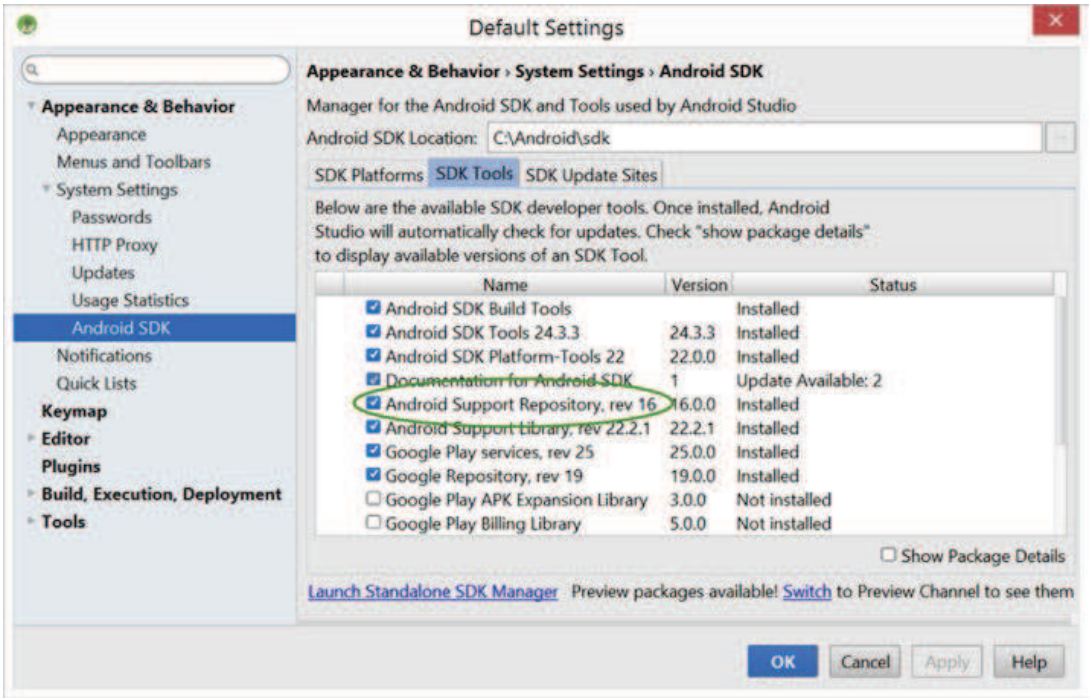

***Figure 1-8.** Android SDK Manager*

By default, Android Studio creates a `src/androidTest/java` folder when you create the project using the project wizard so you shouldn't have to create any new directory. If you can't see it, then check that the Test Artifact in the Build Variant window is set to Android Instrumentation Tests (see Figure 1-9).

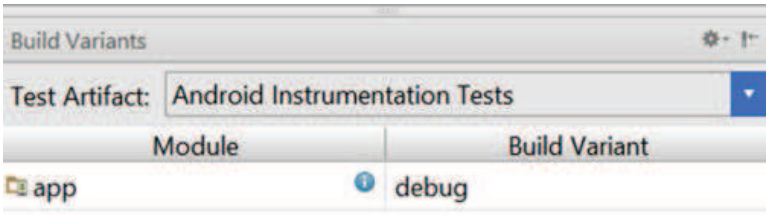

Figure 1-9. Build Variant test artifacts

Add the following Espresso libraries (see Listing 1-5) to the `build.gradle` (app) file in the dependencies section and click the Sync Now link. Open the Gradle console as this may take a minute or two.

Listing 1-5. Espresso Libraries

```
dependencies {
    androidTestCompile 'com.android.support.test:testing-support-lib:0.1'
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.0'
}
```

The code in Listing 1-6 shows how we set up and run the GUI test to add 3 + 4 and how we assert that this is 7.0. In order to test Android activities we need to extend the `CalculatorAddTest` with the `ActivityInstrumentationTestCase2` class. This allows you to take control of the activities. We do this in the `setUp()` method using the `getActivity()` call.

Listing 1-6. Adding Two numbers Using Espresso

```
import android.test.ActivityInstrumentationTestCase2;

import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.action.ViewActions.typeText;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.matcher.ViewMatchers.withId;
import static android.support.test.espresso.matcher.ViewMatchers.withText;

public class CalculatorAddTest extends ActivityInstrumentationTestCase2<
CalculatorActivity> {
```

```
    public static final String THREE = "3";
    public static final String FOUR = "4";
    public static final String RESULT = "7.0";

    public CalculatorAddTest() {
        super(CalculatorActivity.class);
    }

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

    public void testCalculatorAdd() {

        onView(withId(R.id.operand_one_edit_text)).perform(typeText(THREE));
        onView(withId(R.id.operand_two_edit_text)).perform(typeText(FOUR));
        onView(withId(R.id.operation_add_btn)).perform(click());
        onView(withId(R.id.operation_result_text_view)).check(matches
        (withText(RESULT)));
   }
}
```

In the code we first connect to the Calculator Activity and then use the ViewMatcher and ViewActions to put the numbers 3 and 4 in the correct text fields. The code then uses a ViewAction to click the Add button and finally we use the ViewAssertion to make sure the answer is the expected 7.0. Note that the GUI displays the result as a double, so it's 7.0 and not 7 as you might expect (see Figure 1-10).