Understanding Testing with the Android SDK

Contents

- Common assertions
- View assertions
- Other assertion types
- Helpers to test User Interfaces
- Mock objects
- Instrumentation
- TestCase class hierarchies
- Using external libraries

Assertions in depth

- Assertions are methods that check for a condition that can be evaluated.
- They can be grouped together in the following different sets, depending on the condition checked, for example:
  - assertEquals
  - assertTrue
  - assertFalse
  - assertNull
  - assertNotNull
  - assertSame
  - assertNotSame
  - fail

Custom messages

- It is worth knowing that all assert methods provide an overloaded version including a custom String message.
- The premise behind this is that, sometimes, the generic error message does not reveal enough details, and it is not obvious how the test failed.
- This custom message can be extremely useful to easily identify the failure once you are looking at the test report.

example

```
public void testMax() {
int a = 10;
int b = 20;
int actual = Math.max(a, b);
String failMsg = "Expected: " + b + " but was: " +
actual;
assertEquals(failMsg, b, actual);
}
```

Static imports

- Though basic assertion methods are inherited from the Assert base class, some other assertions need specific imports.
- To improve the readability of your tests, there is a pattern to statically import the assert methods from the corresponding classes.

example

Using this pattern instead of having:

public void testAlignment() {

int margin = 0;

...

android.test.ViewAsserts.assertRightAligned(errorMsg, editText, margin);

}

We can simplify it by adding the static import:

import static android.test.ViewAsserts.assertRightAligned;

public void testAlignment() {

int margin = 0;

assertRightAligned(errorMsg, editText, margin);

}

View assertions

- The assertions introduced earlier handle a variety of types as parameters, but they are only intended to test simple conditions or simple objects.
- Android provides a class with plenty of assertions in android.test.ViewAsserts which test relationships between Views and their absolute and relative positions on the screen.

example

The following example shows how you can use ViewAssertions to test the user interface

layout:

```
public void testUserInterfaceLayout() {
int margin = 0;
View origin = mActivity.getWindow().getDecorView();
assertOnScreen(origin, editText);
assertOnScreen(origin, button);
assertRightAligned(editText, button, margin);
}
```

Even more assertions

- If the assertions that are reviewed previously do not seem to be enough for your tests' needs, there is still another class included in the Android framework that covers other cases. This class is MoreAsserts

The TouchUtils class

- Sometimes, when testing UIs, it is helpful to simulate different kinds of touch events.These touch events can be generated in many different ways, but probably android.test.TouchUtils is the simplest to use.
- The TouchUtils class provides the infrastructure to inject the events using the correct UI or main thread, so no special handling is needed, and you don’t need to annotate the test using @UIThreadTest.TouchUtils supports the following:
  - Clicking on a View and releasing it
  - Tapping on a View (touching it and quickly releasing)
  - Long-clicking on a View
  - Dragging the screen
  - Dragging Views

Mock objects

- The Android SDK provides the following classes in the subpackage android.test.mock to help us:
  - MockApplication: This is a mock implementation of the Application class.
  - MockContentProvider: This is a mock implementation of ContentProvider.
  - MockContentResolver: This is a mock implementation of the ContentResolver class that isolates the test code from the real content system.
  - MockContext: This is a mock context class, and this can be used to inject other dependencies.
  - MockCursor: This is a mock Cursor class that isolates the test code from real Cursor implementation.
  - MockDialogInterface: This is a mock implementation of the DialogInterface class.
  - MockPackageManager: This is a mock implementation of the PackageManager class.
  - MockResources: This is a mock Resources class.

.

The IsolatedContext class

- In your tests, you might find the need to isolate the Activity under test from other Android components to prevent unwanted interactions.
- For those cases, the Android SDK provides android.test.IsolatedContext, a mock Context that not only prevents interaction with most of the underlying system but also satisfies the needs of interacting with other packages or components such as Services or ContentProviders.

Alternate route to file and database operations

- In some cases, all we need is to be able to provide an alternate route to the file and database operations. For example, if we are testing the application on a real device, we perhaps don’t want to affect the existing database but use our own testing data.
- Such cases can take advantage of another class that is not part of the android.test.mock subpackage but is part of android.test instead, that is, RenamingDelegatingContext.

The MockContentResolver class

- The MockContentResolver class implements all methods in a non-functional way and throws the exception UnsupportedOperationException if you attempt to use them.

The TestCase base class

- This is the base class of all other test cases in the JUnit framework. It implements the basic methods that we were analyzing in the previous examples (setUp()). Your Android test cases should always extend TestCase or one of its descendants

The default constructor

- All test cases require a default constructor because, sometimes, depending on the test runner used, this is the only constructor that is invoked, and is also used for serialization.
- According to the documentation, this method is not intended to be used by “mere mortals” without calling setName(String name).

example

- Therefore, to appease the Gods, a common pattern is to use a default test case name in this constructor and invoke the given name constructor afterwards:

```
public class MyTestCase extends TestCase {
public MyTestCase() {
this("MyTestCase Default Name");
}
public MyTestCase(String name) {
super(name);
}
}
```

The given name constructor

- This constructor takes a name as an argument to label the test case. It will appear in test reports and would be of much help when you try to identify where failed tests have come from.

The setName() method

- There are some classes that extend TestCase that don’t provide a given name constructor.
- In such cases, the only alternative is to call setName(String name).

The AndroidTestCase base class

- This class can be used as a base class for general-purpose Android test cases.
- Use it when you need access to Android resources, databases, or files in the filesystem.
- Context is stored as a field in this class, which is conveniently named mContext and can be used inside the tests if needed, or the getContext() method can be used too.
- Tests based on this class can start more than one Activity using Context.startActivity().

- There are various test cases in Android SDK that extend this base class:
  - ApplicationTestCase<T extends Application>
  - ProviderTestCase2<T extends ContentProvider>
  - ServiceTestCase<T extends Service>

assertActivityRequiresPermission() method

- This assertion method checks whether the launching of a particular Activity is protected by a specific permission. It takes the following three parameters:
  - packageName: This is a string that indicates the package name of the activity to
  - launch
  - className: This is a string that indicates the class of the activity to launch
  - permission: This is a string with the permission to check

example

- This test checks the requirement of the android.Manifest.permission.WRITE_EXTERNAL_STORAGE permission, which is needed to write to external storage, in the MyContactsActivity Activity:

```
public void testActivityPermission() {
String pkg = "com.blundell.tut";
String activity = PKG + ".MyContactsActivity";
String permission = android.Manifest.permission.CALL_PHONE;
assertActivityRequiresPermission(pkg, activity, permission);
}
```

The assertReadingContentUriRequiresPermission method

- This assertion method checks whether reading from a specific URI requires the permission provided as a parameter.
- It takes the following two parameters:
  - uri: This is the Uri that requires a permission to query
  - permission: This is a string that contains the permission to query
- If a SecurityException class is generated, which contains the specified permission, this assertion is validated.

Example

- This test tries to read contacts and verifies that the correct SecurityException is generated:

```
public void testReadingContacts() {
Uri URI = ContactsContract.AUTHORITY_URI;
String PERMISSION =
android.Manifest.permission.READ_CONTACTS;
assertReadingContentUriRequiresPermission(URI,
PERMISSION);
}
```

The assertWritingContentUriRequiresPermission() method

- The signature for this method is as follows:

public void assertWritingContentUriRequiresPermission (Uri uri, String permission)

Description

- This assertion method checks whether inserting into a specific Uri requires the permission provided as a parameter.
- It takes the following two parameters:
  - uri: This is the Uri that requires a permission to query
  - permission: This is a string that contains the permission to query
- If a SecurityException class is generated, which contains the specified permission, this assertion is validated.

example

- This test tries to write to Contacts and verifies that the correct SecurityException is generated:

```
public void testWritingContacts() {
Uri uri = ContactsContract.AUTHORITY_URI;
String permission =
android.Manifest.permission.WRITE_CONTACTS;
assertWritingContentUriRequiresPermission(uri,
permission);
}
```

Instrumentation

- Instrumentation is instantiated by the system before any of the application code is run, thereby allowing monitoring of all the interactions between the system and the application.

The ActivityMonitor inner class

- The Instrumentation class is used to monitor the interaction between the system and the application or the Activities under test.
- The inner class Instrumentation ActivityMonitor allows the monitoring of a single Activity within an application.

The InstrumentationTestCase class

- The InstrumentationTestCase class is the direct or indirect base class for various test cases that have access to Instrumentation. This is the list of the most important direct and indirect subclasses:
  - ActivityTestCase
  - ProviderTestCase2<T extends ContentProvider>
  - SingleLaunchActivityTestCase<T extends Activity>
  - SyncBaseInstrumentation
  - ActivityInstrumentationTestCase2<T extends Activity>
  - ActivityUnitTestCase<T extends Activity>
- The InstrumentationTestCase class is in the android.test package, and extends junit.framework.TestCase, which extends junit.framework.Assert.

The launchActivity and launchActivityWithIntent methods

- These utility methods are used to launch Activities from a test. If the Intent is not specified using the second option, a default Intent is used:

public final T launchActivity (String pkg, Class<T> activityCls, Bundle extras)

The sendKeys and sendRepeatedKeys methods

- While testing Activities UI, we face the need to simulate interaction with qwertybased keyboards or DPAD buttons to send keys to complete fields, select shortcuts, or navigate throughout the different components.
- There is one version of sendKeys that accepts integer keys values. They can be obtained from constants defined in the KeyEvent class.

The runTestOnUiThread helper method

The runTestOnUiThread method is a helper method used to run portions of a test on the

UI thread. We used this inside the method requestMessageInputFocus(); so that we can

set the focus on our EditText before waiting for the application to be idle, using

Instrumentation.waitForIdleSync().

The ActivityTestCase class

- This is mainly a class that holds common code for other test cases that access Instrumentation.
- You can use this class if you are implementing a specific behavior for test cases and the existing alternatives don’t fit your requirements.
- If this is not the case, you might find the following options more suitable for your requirements:
  - ActivityInstrumentationTestCase2<T extends Activity>
  - ActivityUnitTestCase<T extends Activity>

The scrubClass method

- The scrubClass method is one of the protected methods in the class:

protected void scrubClass(Class<?> testCaseClass)

- It is invoked from the tearDown() method in several of the discussed test case implementations in order to clean up class variables that may have been instantiated as non-static inner classes so as to avoid holding references to them.
- This is in order to prevent memory leaks for large test suites.
- IllegalAccessException is thrown if a problem is encountered while accessing these class variables.

The ActivityInstrumentationTestCase2 class

- The ActivityInstrumentationTestCase2 class would probably be the one you use the most to write functional Android test cases. It provides functional testing of a single Activity.
- This class has access to Instrumentation and will create the Activity under test using the system infrastructure, by calling InstrumentationTestCase.launchActivity().
- If you need to provide a custom Intent to start your Activity, before invoking getActivity(), you may inject an Intent with setActivityIntent(Intent intent).

The constructor

- There is only one public non-deprecated constructor for this class, which is as follows:
- ActivityInstrumentationTestCase2(Class<T> activityClass)
- It should be invoked with an instance of the Activity class for the same Activity used as a class template parameter.

The setUp method

- The *setUp* method is the precise place to initialize the test case fields and other fixture components that require initialization.

example

- This is an example that shows some of the patterns that you might repeatedly find in your test cases:

```
@Override
protected void setUp() throws Exception {
super.setUp();
// this must be called before getActivity()
// disabling touch mode allows for sending key events
setActivityInitialTouchMode(false);
activity = getActivity();
instrumentation = getInstrumentation();
linkTextView = (TextView) activity.findViewById(R.id.main_text_link);
messageInput = (EditText)
activity.findViewById(R.id.main_input_message);
capitalizeButton = (Button)
activity.findViewById(R.id.main_button_capitalize);
}
```

The tearDown method

- Usually, this method cleans up what was initialized in setUp. For instance, if you were creating an integration test that sets up a mock web server before your tests, you would want to tear it back down afterwards to free up resources.
- In this example, we ensure that the object we used is disposed of:

```
@Override
protected void tearDown() throws Exception {
super.tearDown();
myObject.dispose();
}
```

The ProviderTestCase2<T> class

- This is a test case designed to test the ContentProvider classes.
- The ProviderTestCase2 class also extends AndroidTestCase. The class template parameter T represents ContentProvider under test. Implementation of this test uses
- IsolatedContext and MockContentResolver, which are mock objects that we described before in this chapter.

The constructor

- There is only one public non-deprecated constructor for this class. This is as follows:
- ProviderTestCase2(Class<T> providerClass, String providerAuthority)
- This should be invoked with an instance of the ContentProvider class for the same ContentProvider class used as a class template parameter.
- The second parameter is the authority for the provider, which is usually defined as the AUTHORITY constant in the ContentProvider class.

An example

- This is a typical example of a ContentProvider test:

```
public void testQuery() {
String segment = "dummySegment";
Uri uri = Uri.withAppendedPath(MyProvider.CONTENT_URI, segment);
Cursor c = provider.query(uri, null, null, null, null);
try {
int actual = c.getCount();
assertEquals(2, actual);
} finally {
c.close();
}
}
```

The ServiceTestCase<T>

- This is a test case specially created to test services. The methods to exercise the service life cycle, such as setupService, startService, bindService, and shutDownService, are also included in this class.

The constructor

- There is only one public non-deprecated constructor for this class. This is as follows:
- ServiceTestCase(Class<T> serviceClass)
- It should be invoked with an instance of the Service class for the same Service used as a class template parameter.

The TestSuiteBuilder.FailedToCreateTests class

- The TestSuiteBuilder.FailedToCreateTests class is a special TestCase class used to indicate a failure during the build() step.
- That is, during the test suite creation, if an error is detected, you will receive an exception, which indicates the failure to construct the test suite:

Using libraries in test projects

- Your Android project might require an external Java library or an Android library.
- The following explains the usage of a local module that is an Android library, but the same rules can be applied to an external JAR (Java library) file or an external AAR (Android library) file.

Continue…

- Let’s pretend that in one Activity, we are creating objects from a class that is part of a library. For the sake of our example, let’s say the library is called dummyLibrary, and the mentioned class is Dummy.
- So our Activity would look like this:

```
import com.blundell.dummylibrary.Dummy;

public class MyFirstProjectActivity extends Activity {
    private Dummy dummy;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        final EditText messageInput = (EditText)
findViewById(R.id.main_input_message);
        Button capitalizeButton = (Button)
findViewById(R.id.main_button_capitalize);
        capitalizeButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                String input = messageInput.getText().toString();
                messageInput.setText(input.toUpperCase());
            }
        });

        dummy = new Dummy();
    }

    public Dummy getDummy() {
        return dummy;
    }

    public static void methodThatShouldThrowException() throws Exception {
        throw new Exception("This is an exception");
    }

}
```

- This library is an Android AAR module, and so it should be added to your build.gradle dependencies in the normal way:

```
dependencies {
compile project(':dummylibrary')
}
```

- If this was an external library, you would replace project(':dummylibrary') with 'com.external.lib:name:version'.
- Now, let’s create a simple test. From our previous experience, we know that if we need to test an Activity, we should use ActivityInstrumentationTestCase2, and this is precisely what we will do. Our simple test will be as follows:

```
public void testDummy() {
assertNotNull(activity.getDummy());
}
```

Summary

- We investigated the most relevant building blocks and reusable patterns to create our tests.
- Along this journey, we:
  - Understood the common assertions found in JUnit tests
  - Explained the specialized assertions found in the Android SDK
  - Explored Android mock objects and their use in Android tests
  - Exemplified the use of the different test cases available in the Android SDK