Lesson 6: Testing your UI

6.1: UI testing

Contents:

- Properly testing a user interface (UI)
- Setting up your test environment
- Using Espresso for tests that span a single app
- Using UI Automator for tests that span multiple apps
- Related practical
- Learn more

Properly testing a user interface (UI)

Writing and running tests is an important part of the Android app-development cycle. Well-written tests can help you catch bugs early in your development cycle, where bugs are easier to fix. Tests also help you gain confidence in your code.

In *user interface (UI) testing*, you focus on aspects of the UI and the app's interactions with users. Recognizing and acting on user input is a high priority in UI testing and validation. You need to make sure that your app not only recognizes the type of input but also acts accordingly.

As a developer, you should get into the habit of testing user interactions to ensure that users don't encounter unexpected results or have a poor experience when interacting with your app. UI testing can help you recognize the input controls where unexpected input should be handled gracefully or should trigger input validation.

Note: We strongly encourage you to use Android Studio for building your test apps, because it provides project setup, library inclusion, and packaging conveniences. You can run UI tests on a variety of physical or virtual Android devices, and you can then analyze the results and make changes to your code without leaving the development environment.

An app's UI contains `View` elements such as buttons, menus, and text fields, each with a set of properties. To properly test a UI, you need to do the following:

- Exercise all UI events that use `View` elements. To do this, tap a `View` in the app and enter data or make a choice. Then examine the *values of the properties* of each `View`—referred to as the *state* of the UI—at different times during execution.
- Provide inputs to all UI `View` elements. Use this opportunity to test improper inputs. For example, input letters in a view where numbers are expected.
- Check the outputs and UI representations of data—such as strings and integers—to see if they are consistent with what you expect.

In addition to functionality, UI testing evaluates design elements such as layout, colors, fonts, font sizes, labels, text boxes, text formatting, captions, buttons, lists, icons, links, and content.

Manual testing

As the developer of an app, you probably test each UI component manually as you add the component to the app's UI. As development continues, one approach to UI testing is to simply have a human tester perform a set of user operations on the target app and verify that it is behaving correctly.

However, this manual approach can be time-consuming, tedious, and error-prone. By manually testing a UI for a complex app, you can't possibly cover all permutations of user interactions. You would also have to manually perform these repetitive tests on many different device configurations in an emulator, and on many different devices. To summarize, the problems inherent with manual testing fall into two categories:

- *Domain size:* A UI has many operations that need to be tested. Even a relatively small app can have hundreds of possible UI operations. Over a development cycle a UI may change significantly, even though the underlying app doesn't change. Manual tests with instructions to follow a certain path through the UI may fail over time, because a button, menu item, or dialog may have changed location or appearance.
- *Sequences:* Some functionality of the app may only be accomplished with a sequence of UI events. For example, to add an image to a message about to be sent, a user may have to tap a camera button and use the camera to take a picture, or a photo button to select an existing picture, and then associate that picture with the message—usually by tapping a share or send button. Increasing the number of possible operations also increases the sequencing problem.

Automated testing

When you automate tests of user interactions, you free yourself and your resources for other work. To generate a set of test cases, test designers attempt to cover all of the functionality of the system and fully exercise the UI. Performing all of the UI interactions automatically makes it easier to run tests for different device states (such as orientations) and different configurations.

For testing Android apps, you typically create these types of automated UI tests:

- *UI tests that work within a single app:* Verifies that the app behaves as expected when a user performs a specific action or enters a specific input. It allows you to check that the app returns the correct UI output in response to user interactions in each app `Activity`. UI testing frameworks like Espresso allow you to programmatically simulate user actions and test complex intra-app user interactions.
- *UI tests that span multiple apps:* Verifies the correct behavior of interactions between different user apps or between user apps and system apps. For example, you can test an app that launches the Maps app to show directions, or that launches the Android contact picker to select recipients for a message. UI testing frameworks that support cross-app interactions, such as UI Automator, allow you to create tests for such user-driven scenarios.

Using Espresso for tests that span a single app

The Espresso testing framework, in the Android Testing Support Library, provides APIs for writing UI tests to simulate user interactions within a single app. Espresso tests run on actual device or emulator and behave as if an actual user is using the app.

You can use Espresso to create UI tests to automatically verify the following:

- The app returns the correct UI output in response to a sequence of user actions on a device.
- The app's navigation and input controls bring up the correct `Activity` and `View` elements.
- The app responds correctly with mocked-up dependencies, such as data from an outside server, or can work with stubbed out backend methods to simulate real interactions with backend components which can be programmed to reply with a set of defined responses.

A key benefit of using Espresso is that it has access to instrumentation information, such as the context for the app, so that you can monitor all of the interaction the Android system has with the app. Another key benefit is that it automatically synchronizes test actions with the app's UI. Espresso detects when the main thread is idle, so it is able to run your test at the appropriate time, improving the reliability of your tests. This capability also relieves you from having to add any timing workarounds, such as a sleep period, in your test code.

The Espresso testing framework works with the AndroidJUnitRunner test runner and requires instrumentation, which is described later in this section. Espresso tests can run on devices running Android 2.2 (API level 8) and higher.

Using UI Automator for tests that span multiple apps

The UI Automator testing framework in the Android Testing Support Library can help you verify the correct behavior of interactions between different user apps or between user apps and system apps. It can also show you what is happening on the device before and after an app is launched.

The UI Automator APIs let you interact with visible elements on a device. Your test can look up a UI element by using descriptors such as the text displayed in that element or its content description. A viewer tool provides a visual interface to inspect the layout hierarchy and view the properties of UI elements that are visible on the foreground of the device.

The following are important functions of UI Automator:

- Like Espresso, UI Automator has access to system interaction information so that you can monitor all of the interaction the Android system has with the app.
- Your test can send an `Intent` or launch an `Activity` (without using shell commands) by getting a `Context` object through `getContext()`.
- You can simulate user interactions on a collection of items, such as songs in a music album or a list of emails in an inbox.
- You can simulate vertical or horizontal scrolling across a display.
- You can use standard JUnit `Assert` methods to test that UI components in the app return the expected results.

The UI Automator testing framework works with the AndroidJUnitRunner test runner and requires instrumentation, which is described in the next section. UI Automator tests can run on devices running Android 4.3 (API level 18) or higher.

What is instrumentation?

Android instrumentation is a set of control methods, or *hooks*, in the Android system, which control Android components and how the Android system loads apps.

Normally the system runs all the components of an app in the same process. You can allow some components, such as content providers, to run in a separate process, but you typically can't force an app onto the same process as another running app.

Instrumentation tests, however, can load both a test package and the app into the same process. Since the app components and their tests are in the same process, your tests can invoke methods in the components, and modify and examine fields in the components.

Instrumentation allows you to monitor all of the interaction the Android system has with the application, and makes it possible for tests to invoke methods in the app, and modify and examine fields in the app, independently of the app's normal lifecycle.

Normally, an Android component runs in a lifecycle that the system determines. For example, an `Activity` object's lifecycle starts when an `Intent` activates the `Activity`. The system calls the object's `onCreate()` method, and then the `onResume()` method. When the user starts another app, the system calls the `onPause()` method. If the `Activity` code calls the `finish()` method, the system calls the `onDestroy()` method. The Android framework API does not provide a way for your app's code to invoke these callback methods directly, but you can do so using an Espresso or UI Automator test with instrumentation.

Setting up your test environment

To use the Espresso and UI Automator frameworks, you need to store the source files for instrumented tests at *module-name*`/src/androidTests/java/` (in which *module-name* is the name of the app or module).
This directory already exists when you create a new Android Studio project. In the **Project > Android** view of Android Studio, show this directory by navigating to **app > java > *module-name* (androidTest)**.

You also need to do the following:

- Install the Android Support Repository and the Android Testing Support Library.
- Add dependencies to the project's `build.gradle (Module: app)` file.
- Create test files in the `androidTest` directory.

Installing the Android Support Repository and Testing Support Library

You may already have the Android Support Repository and its Android Testing Support Library installed with Android Studio. To check for the Android Support Repository, follow these steps:

1. In Android Studio select **Tools > Android > SDK Manager**.
2. Click the **SDK Tools** tab and look for the Support Repository.
3. If necessary, update or install the library.

Adding the dependencies

When you start a project for the Phone and Tablet form factor using **API 15: Android 4.0.3 (Ice Cream Sandwich)** as the minimum SDK, Android Studio automatically includes the dependencies you need to use Espresso.

Note: If you created your project in a previous version of Android Studio, you may have to add the dependencies and instrumentation statement yourself.

To ensure that you have these dependencies, follow these steps:

1. Open the **build.gradle (Module: app)** file.
2. Check if the following is included (along with other dependencies) in the `dependencies` section:

```
3.  testImplementation 'junit:junit:4.12'
4.  androidTestImplementation 'com.android.support.test:runner:1.0.1'
5.  androidTestImplementation
6.              'com.android.support.test.espresso:espresso-core:3.0.1'
```

   If the file doesn't include the above dependency statements, enter them into the `dependencies` section.
7. Android Studio also adds the following instrumentation statement to the end of the `defaultConfig` section of a new project:

```
8.  testInstrumentationRunner
9.                  "android.support.test.runner.AndroidJUnitRunner"
```

   If the file doesn't include the above instrumentation statement, enter it at the end of the `defaultConfig` section.
10. If you modified the **build.gradle (Module: app)** file, click the **Sync Now** link in the notification about Gradle files in top right corner of the window.

Setting up the test rules and annotations

To write tests, Espresso and UI Automator use JUnit as their testing framework. JUnit is the most popular and widely-used unit testing framework for Java. Your test class using Espresso or UI Automator should be written as a JUnit 4 test class.

Note: The most current JUnit revision is JUnit 5. However for the purposes of using Espresso or UI Automator, version 4.12 is recommended.

To create a basic JUnit 4 test class, create a Java class for testing in the directory specified at the beginning of this section. It should contain one or more methods and behavior rules defined by JUnit annotations.

For example, the following shows a test class definition with annotations:

```
@RunWith(AndroidJUnit4.class)
public class ActivityInputOutputTest {

    @Rule
    public ActivityTestRule mActivityRule = new ActivityTestRule<>(
            MainActivity.class);

    @Test
    public void useAppContext() throws Exception {
        // Context of the app under test.
        Context appContext =
                    InstrumentationRegistry.getTargetContext();

        assertEquals("com.example.android.twoactivities",
                    appContext.getPackageName());
    }

    @Test
    public void activityLaunch() {
        onView(withId(R.id.button_main)).perform(click());
        onView(withId(R.id.text_header))
                            .check(matches(isDisplayed()));
        onView(withId(R.id.button_second)).perform(click());
        onView(withId(R.id.text_header_reply))
                            .check(matches(isDisplayed()));
    }

    // More tests...
}
```

The following annotations are useful for testing:

@RunWith

To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition, which indicates the runner that will be used to run the tests in this class. A test runner is a library or set of tools that enables testing to occur and the results to be printed to a log.

@SmallTest, @MediumTest, and @LargeTest

The `@SmallTest`, `@MediumTest`, and `@LargeTest` Android annotations provide some clarity about what resources and features the test uses. For example, the `@SmallTest` annotation tells you that the test doesn't interact with any file system or network.

The following table summarizes what the annotations mean:

Feature	Small	Medium	Large
Network access	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

For a description of the Android `@SmallTest`, `@MediumTest`, and `@LargeTest` annotations, see Test Sizes in the Google Testing Blog. For a summary of JUnit annotations, see Package org.junit.

@Rule

Before declaring the test methods, use the `@Rule` annotation, such as `ActivityTestRule` or `ServiceTestRule`. The `@Rule` establishes the context for the testing code. For example:

```
@Rule
public ActivityTestRule mActivityRule = new ActivityTestRule<>(
                    MainActivity.class);
```

This rule uses an `ActivityTestRule` object, which provides functional testing of a single `Activity`—in this case, `MainActivity.class`.

`ServiceTestRule` is a JUnit rule that provides a simplified mechanism to start and shut down your service before and after the duration of your test. It also guarantees that the service is successfully connected when starting (or binding to) a service.

@Test

A test method begins with the `@Test` annotation and contains the code to exercise and verify a single function in the component that you want to test:

```
@Test
public void testActivityLaunch() {//...}
```

The `Activity` under test will be launched before each test annotated with `@Test`. During the duration of the test you will be able to manipulate your `Activity` directly.

@Before and @After

In certain rare cases you may need to set up variables or execute a sequence of steps *before* or *after* performing the UI test. You can specify methods to run before running a `@Test` method, using the `@Before` annotation, and methods to run after, using the `@After` annotation.

- `@Before`: The `@Test` method will execute *after* the methods designated by the `@Before` annotation. The `@Before` methods terminate prior to the execution of the `@Test` method.
- `@After`: The `@Test` method will execute *before* the methods designated by the `@After` annotation.

Using Espresso for tests that span a single app

When writing tests it is sometimes difficult to get the balance right between over-specifying the test or not specifying enough. Over-specifying the test can make it brittle to changes, while under-specifying may make the test less valuable, since it continues to pass even when the UI element and its code under test is broken.

To make tests that are balanced, it helps to have a tool that allows you to pick out precisely the aspect under test and describe the values it should have. Such tests fail when the behavior of the aspect under test deviates from the expected behavior, yet continue to pass when minor, unrelated changes to the behavior are made. The tool available for Espresso is the Hamcrest framework.

Hamcrest (an anagram of "matchers") is a framework that assists writing software tests in Java. The framework lets you create custom assertion matchers, allowing match rules to be defined declaratively.

Tip: To learn more about Hamcrest, see The Hamcrest Tutorial.

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Writing Espresso tests with Hamcrest Matchers

You write Espresso tests based on what a user might do while interacting with your app. The key concepts are *locating* and then *interacting* with UI elements. These are the basic steps:

1. **Match a View:** Find a `View`.
2. **Perform an action:** Perform a click or other action that triggers an event with the `View`.
3. **Assert and verify the result:** Check the state of the `View` to see if it reflects the expected state or behavior defined by the assertion.

With Espresso you use the following types of Hamcrest expressions to help find `View` elements and interact with them:

- *ViewMatchers:* Hamcrest matcher expressions in the `ViewMatchers` class that lets you find a `View` in the current `View` hierarchy so that you can examine something or perform some action.
- *ViewActions:* Hamcrest action expressions in the `ViewActions` class that lets you perform an action on a `View` found by a `ViewMatcher`.
- *ViewAssertions:* Hamcrest assertion expressions in the `ViewAssertions` class that lets you assert or check the state of a `View` found by a `ViewMatcher`.

The following shows how all three expressions work together:

1. Use a `ViewMatcher` to find a `View`: `onView(withId(R.id.my_view))`
2. Use a `ViewAction` to perform an action: `.perform(click())`
3. Use a `ViewAssertion` to check if the result of the action matches an assertion: `.check(matches(isDisplayed()))`

The following shows how the above expressions are used together in a statement:

```
onView(withId(R.id.my_view))
        .perform(click())
        .check(matches(isDisplayed()));
```

Use `onView()` with a `ViewMatcher` so that you can examine something or perform some action. The most common ones are:

- `withId()`: Find a view with a specific `android:id` (which is typically defined in the layout XML file). For example:
- `onView(withId(R.id.my_view))`
- `withText()`: Find a `View` with specific text, typically used with `allOf()` and `withId()`. For example, the following uses `allOf` to cause a match if the examined object matches *all* of the specified conditions—the `View` uses the `word` id (`withId`), and the `View` has the text `"Clicked! Word 15"` (`withText`):
- `onView(allOf(withId(R.id.word), withText("Clicked! Word 15"),`
- `                                        isDisplayed()))`
- Others including matchers for state (`selected`, `focused`, `enabled`), and content description and hierarchy (`root` and `children`).

You would typically combine a `ViewMatcher` and a `ViewAction` in a single statement, followed by a `ViewAssertion` expression in a separate statement or included in the same statement.
You can see how all three expressions work in the following statement, which combines a `ViewMatcher` to find a view, a `ViewAction` to perform an action, and a `ViewAssertion` to check if the result of the action matches an assertion:

```
// withId(R.id.my_view) is a ViewMatcher
onView(withId(R.id.my_view))
    // click() is a ViewAction
    .perform(click())
    // matches(isDisplayed()) is a ViewAssertion
    .check(matches(isDisplayed()));
```

Why is the Hamcrest framework useful for tests? A simple assertion, such as `assert (x == y)`, lets you assert during a test that a particular condition must be true. If the condition is false, the test fails. But the simple assertion provides no useful error message. With a family of assertions, you can produce more useful error messages, but this leads to an explosion in the number of assertions.
With the Hamcrest framework, it is possible to define operations that take matchers as arguments and return them as results, leading to a grammar that can generate a huge number of possible matcher expressions from a small number of primitive matchers.

For a Hamcrest tutorial, see The Hamcrest Tutorial. For a quick summary of Hamcrest matcher expressions, see the Espresso cheat sheet.

Testing an AdapterView

In an `AdapterView` such as a `Spinner`, the `View` is dynamically populated with child `View` elements at runtime. If the target `View` you want to test is inside a spinner, the `onView()` method might not work because only a subset of the `View` elements may be loaded in the current view hierarchy.
Espresso handles this by providing a separate `onData()` method, which is able to first load the adapter item and bring it into focus prior to operating on it or any of its children views. The `onData()` method uses a `DataInteraction` object and its methods, such as `atPosition()`, `check()`, and `perform()` to access the target `View`. Espresso handles loading the target `View` element into the current `View` hierarchy, scrolling to the target child `View`, and putting that `View` into focus.
For example, the following `onView()` and `onData()` statements test a `Spinner` item click:

1. Find and click the `Spinner` itself (the test must first click the `Spinner` itself in order to click any item in the `Spinner`):

```
2.  onView(withId(R.id.spinner_simple)).perform(click());
```

3. Find and then click the item in the `Spinner` that matches *all* of the following conditions:
   - An item that is a `String`
   - An item that is equal to the String `"Americano"`

```
onData(allOf(is(instanceOf(String.class)),
                        is("Americano"))).perform(click());
```

As you can see in the above statement, matcher expressions can be combined to create flexible expressions of intent:

- `allOf`: Causes a match if the examined object matches *all* of the specified matchers. You can use `allOf()` to combine multiple matchers, such as `containsString()` and `instanceOf()`.
- `is`: Hamcrest strives to make your tests as readable as possible. The `is` matcher is a wrapper that doesn't add any extra behavior to the underlying matcher, but makes your test code more readable.
- `instanceOf`: Causes a match when the examined object is an instance of the specified type; in this case, a string. This match is determined by calling the `Class.isInstance(Object)` method, passing the object to examine.

The following example illustrates how you would test a `Spinner` using a combination of `onView()` and `onData()` methods:

```
@RunWith(AndroidJUnit4.class)
public class SpinnerSelectionTest {
    @Rule
    public ActivityTestRule mActivityRule = new ActivityTestRule<>(
            MainActivity.class);

    @Test
    public void useAppContext() throws Exception {
        // Context of the app under test.
        Context appContext =
                InstrumentationRegistry.getTargetContext();
        assertEquals("com.example.android.phonenumberspinner",
                appContext.getPackageName());
    }

    /**
     * Iterate through the spinner, selecting each item and
     * checking to see if it matches the string in the array.
     */
    @Test
    public void iterateSpinnerItems() {
        String[] myArray = mActivityRule.getActivity().getResources()
                .getStringArray(R.array.labels_array);
        // Iterate through the spinner array of items.
        int size = myArray.length;
        for (int i=0; i<size; i++) {
            // Find the spinner and click on it.
            onView(withId(R.id.label_spinner)).perform(click());
            // Find the spinner item and click on it.
            onData(is(myArray[i])).perform(click());
            // Find the text view and check that the spinner item
            // is part of the string.
            onView(withId(R.id.text_phonelabel))
              .check(matches(withText(containsString(myArray[i]))));
        }
    }
}
```

The test clicks each `Spinner` item from top to bottom, checking to see if the item appears in the text field. It doesn't matter how many `Spinner` items are defined in the array, or what language is used for the `Spinner` items—the test performs all of them and checks their output against the array.

The following is a step-by-step description of the above test:

1. The `iterateSpinnerItems()` method begins by getting the array used for the spinner items:

```
 public void iterateSpinnerItems() {
    String[] myArray =
          mActivityRule.getActivity().getResources()
          .getStringArray(R.array.labels_array);
```

In the statement above, the test accesses the array (with the id `labels_array`) by establishing the `context` with the `getActivity()` method of the `ActivityTestRule` class, and getting a resources instance in the app's package using `getResources()`.

6. Using the length (`size`) of the array, the `for` loop iterates through each `Spinner` item.

```
 int size = myArray.length;
 for (int i=0; i<size; i++) {
    // Find the spinner and click on it.
```

10. The `onView()` statement within the `for` loop finds the `Spinner` and clicks on it. The test must click the `Spinner` itself in order to click any item in the `Spinner`:

```
// Find the spinner and click on it.
onView(withId(R.id.label_spinner)).perform(click());
```

13. The `onData()` statement finds and clicks a spinner item:

```
// Find the spinner item and click on it.
onData(is(myArray[i])).perform(click());
```

The `Spinner` is populated from the `myArray` array, so `myArray[i]` represents a `Spinner` element from the array. As the `for` loop iterates, it performs a click on each `Spinner` element (`myArray[i]`) it finds.

16. The last `onView()` statement finds the `TextView` (`text_phonelabel`) and checks that the `Spinner` item is part of the string:

```
onView(withId(R.id.text_phonelabel))
        .check(matches(withText(containsString(myArray[i]))));
```

Recording a test

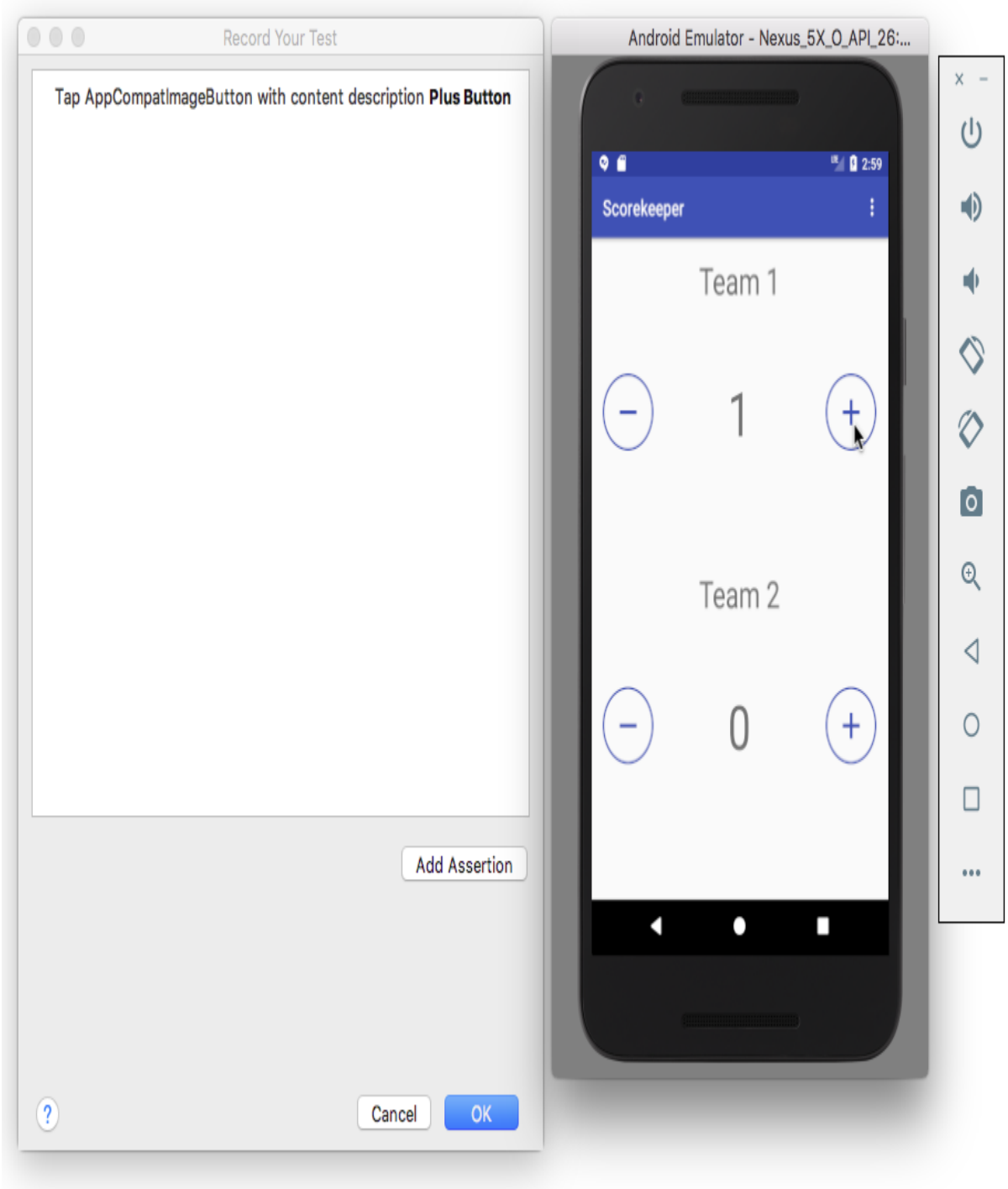
An Android Studio feature (in version 2.2 and newer) lets you *record* an Espresso test, creating the test automatically.

After choosing to record a test, use your app as a normal user would. As you click through the app UI, editable test code is generated for you. Add assertions to check if a `View` holds a certain value.
You can record multiple interactions with the UI in one recording session. You can also record multiple tests, and edit the tests to perform more actions, using the recorded code as a snippet to copy, paste, and edit.

Follow these steps to record a test:

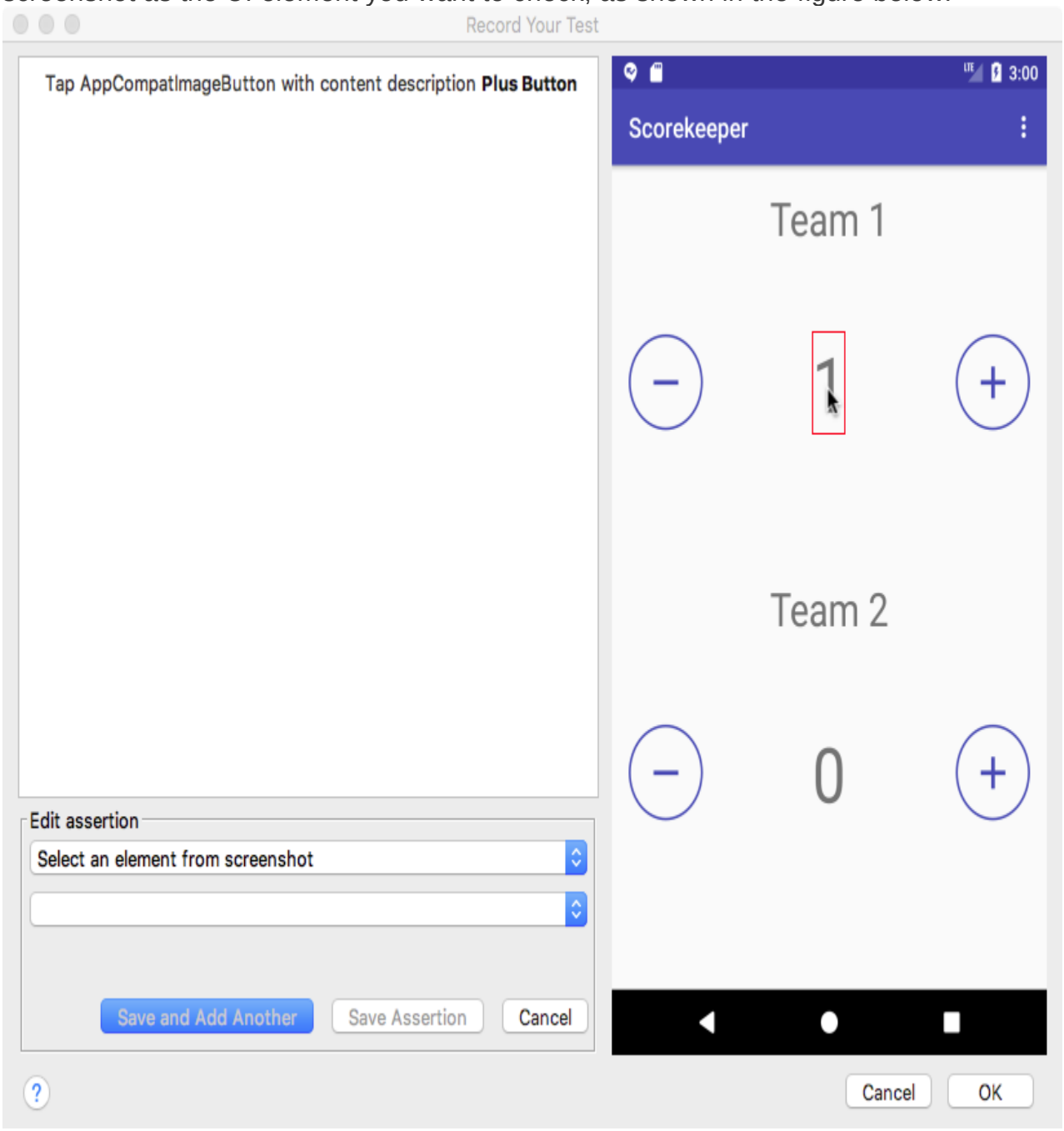
1. Select **Run > Record Espresso Test**, select your deployment target (an emulator or a device), and click **OK**.
2. Interact with the UI to do what you want to test. In this case, tap the plus (**+**) `ImageButton` for Team 1 in the app. The Record Your Test window shows the action that was recorded ("Tap AppCompatImageButton with the content

description **Plus Button**").

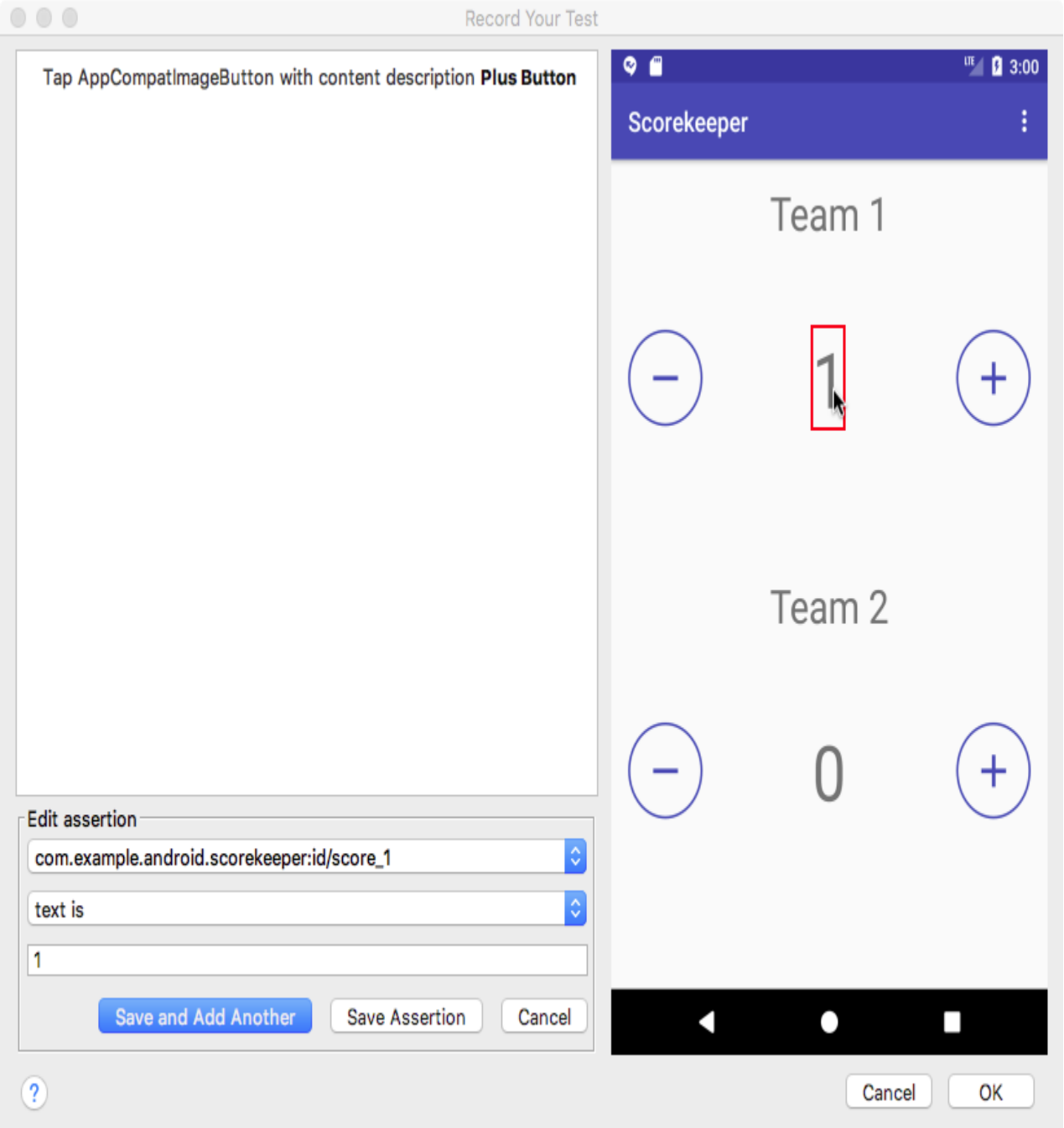


3. Click **Add Assertion** in the Record Your Test window. A screenshot of the app's UI appears in a pane on the right side of the window, and the **Select an element from screenshot** option appears in the dropdown menu. Select the score (**1**) in the

screenshot as the UI element you want to check, as shown in the figure below.



4. Select **text is** from the second dropdown menu, as shown in the figure below. The text you expect to see (**1**) is already entered in the field below the dropdown menu.



5. Click **Save Assertion**, and then click **OK**.

6. In the dialog that appears, edit the name of the test so that it is easy for others to understand the purpose of the test.
7. Android Studio may display a request to add more dependencies to your Gradle Build file. Click **Yes** to add the dependencies. Android Studio adds the following to the dependencies section of the `build.gradle (Module: app)` file:

```
8.  androidTestCompile
```

```
9.            'com.android.support.test.espresso:espresso-contrib:2.2.2', {
10.     exclude group: 'com.android.support', module: 'support-annotations'
11.     exclude group: 'com.android.support', module: 'support-v4'
12.     exclude group: 'com.android.support', module: 'design'
13.     exclude group: 'com.android.support', module: 'recyclerview-v7'
14.  }
```

15. Expand **com.example.android.***appname* **(androidTest)** to see the test, and **right-click** (or **Control-click**) the test to run it.

Using RecyclerViewActions for a RecyclerView

A `RecyclerView` is useful when you have data collections with elements that change at runtime based on user action or network events. `RecyclerView` is a UI component designed to render a collection of data, but is not a subclass of `AdapterView` but of `ViewGroup`. This means that you can't use `onData()`, which is specific to `AdapterView`, to interact with list items.
However, there is a class called `RecyclerViewActions` that exposes an API to operate on a `RecyclerView`. For example, the following test clicks on an item from the list by position:

```
onView(withId(R.id.recyclerView))
    .perform(RecyclerViewActions.actionOnItemAtPosition(0, click()));
```

Using UI Automator for tests that span multiple apps

UI Automator is a set of APIs that can help you verify the correct behavior of interactions between different user apps, or between user apps and system apps. It lets you interact with visible elements on a device. A viewer tool provides a visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device. Like Espresso, UI Automator has access to system interaction information so that you can monitor all of the interaction the Android system has with the app.

To use UI Automator, you must already have set up your test environment in the same way as for Espresso:

- Install the Android Support Repository and the Android Testing Support Library.
- Add the following dependency to the project's `build.gradle (Module: app)` file in the `dependencies` section:

```
androidTestCompile
 'com.android.support.test.uiautomator:uiautomator-v18:2.1.3'
```

- If the `minSdkVersion` is older than 18, switch it to 18. The library works only with version 18 of Android or newer:

Use UI Automator Viewer to inspect the UI on a device

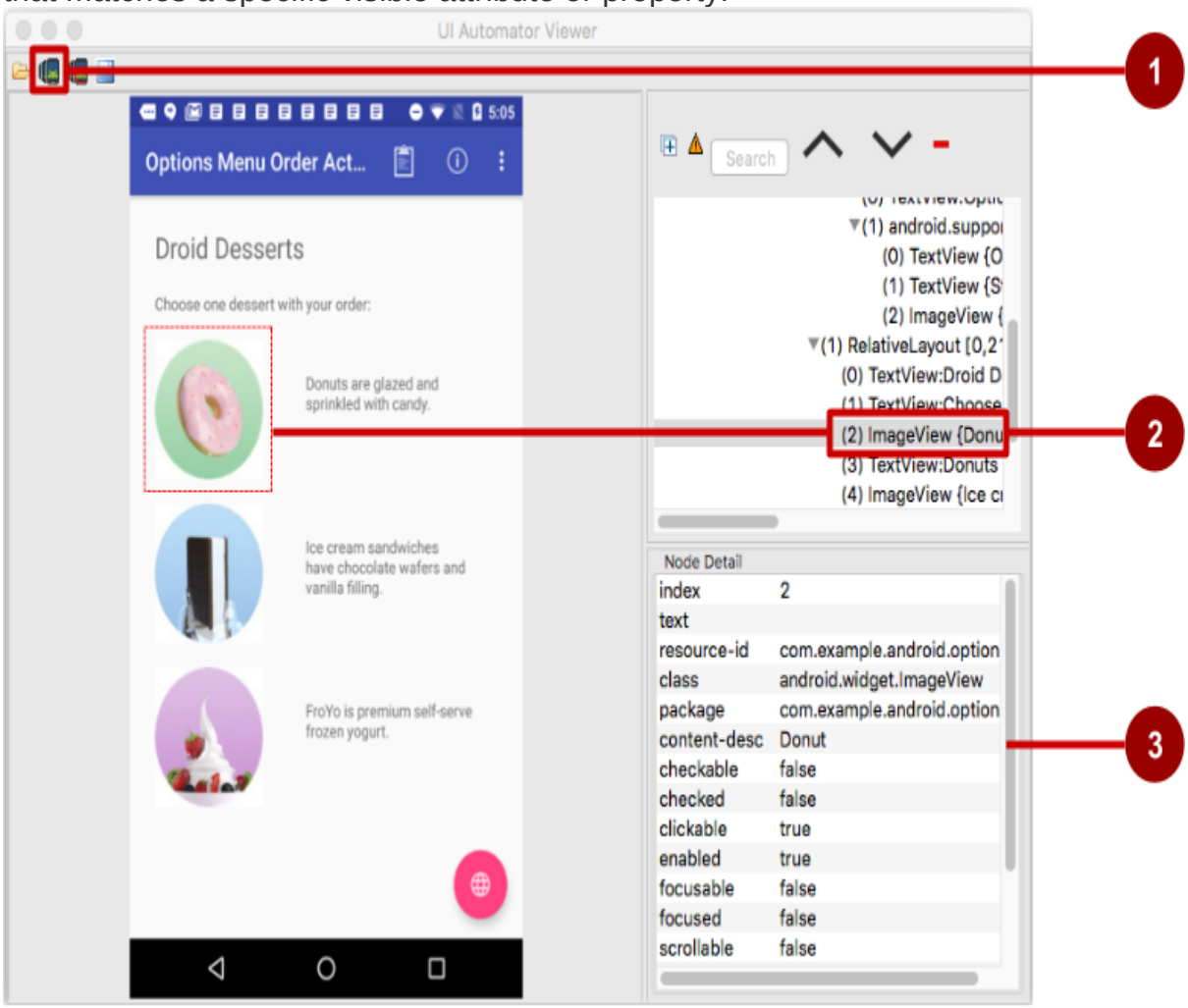
UI Automator Viewer (`uiautomatorviewer`) provides a convenient visual interface to inspect the layout hierarchy and view the properties of UI elements that are visible on the foreground of the device.
To launch the `uiautomatorviewer` tool, follow these steps:

1. Install and then launch the app on a physical device such as a smartphone.
2. Connect the device to your development computer.
3. Open a terminal window and navigate to the **/tools/** directory. To find the specific path, select **Preferences** in Android Studio, and select **Appearance & Behavior > System Settings > Android SDK**. The full path for appears in the Android SDK Location box at the top of the screen.
4. Run the tool with this command: **uiautomatorviewer**

To ensure that your UI Automator tests can access the app's UI elements, check that the elements have visible text labels, `android:contentDescription` values, or both. You can view the properties of a UI element by following these steps (refer to the figure below):

1. After launching `uiautomatorviewer`, the viewer is empty. Click the **Device Screenshot** button.
2. Hover over a UI element in the snapshot in the left-hand panel to see the element in the layout hierarchy in the upper right panel.
3. The layout attributes and other properties for the UI element appear in the lower right panel. Use this information to create tests that select a UI element

that matches a specific visible attribute or property.



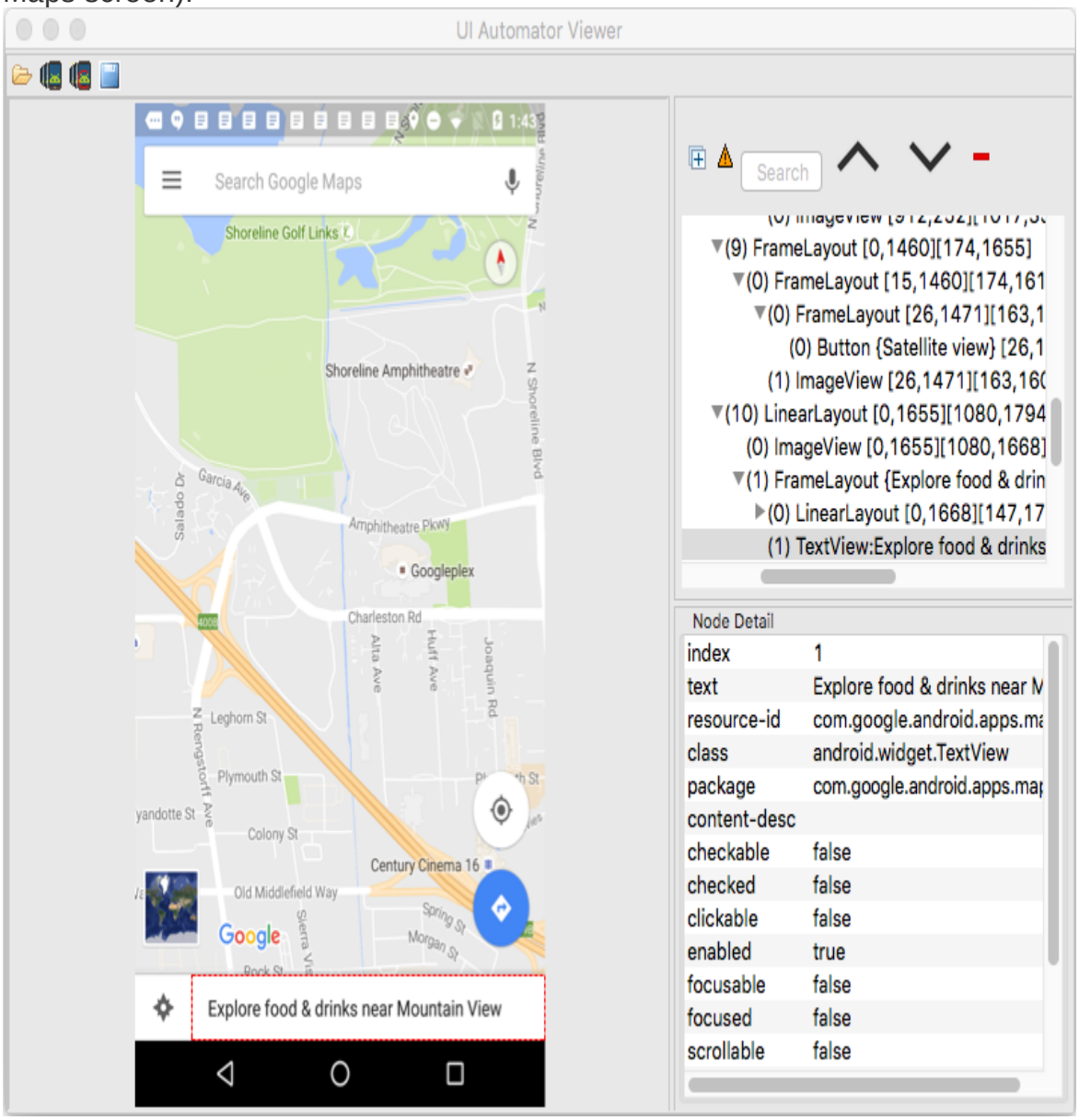
In the above figure:

1. **Device Screenshot** button
2. Selected component in the snapshot and in the layout hierarchy
3. Properties for the selected component

In the app shown above, the red floating action button launches the Maps app. Follow these steps to test the action performed by the floating action button:

1. Tap the floating action button.
2. The code for the button makes an implicit intent to launch the Maps app.

3. Click the **Device Screenshot** button to see the result of the implicit `Intent` (the Maps screen).



Using the viewer, you can determine which UI elements are accessible to the UI Automator framework.

Setup: Ensuring that UI elements are accessible

The UI Automator framework depends on the accessibility features of the Android framework to look up individual UI elements. Implement view properties as follows:

- Include the `android:contentDescription` attribute in your XML layout to label each `ImageButton`, `ImageView`, `CheckBox`, and other user input controls. The following shows the `android:contentDescription` attribute added to a `RadioButton` using the same string resource used for the `android:text` attribute:

```
<RadioButton
 android:id="@+id/sameday"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@string/same_day_messenger_service"
 android:contentDescription=
                        "@string/same_day_messenger_service"
 android:onClick="onRadioButtonClicked"/>
```

You can make input controls more accessible for the sight-impaired by using the `android:contentDescription` XML layout attribute. The text in this attribute does not appear on screen, but if the user enables accessibility services that provide audible prompts, then when the user navigates to that control, the text is spoken.

- Provide an `android:hint` attribute for `EditText` elements (in addition to `android:contentDescription`, which is useful for accessibility services). With `EditText` elements, UI Automator looks for the `android:hint` attribute.
- Associate an `android:hint` attribute with any graphical icons used by controls that provide feedback to the user (for example, status or state information).

As a developer, you should implement the above minimum optimizations to support your users as well as UI Automator.

Creating a test class

A UI Automator test class generally follows this programming model:

1. **Access the device to test:** An instance of the `InstrumentRegistry` class holds a reference to the instrumentation running in the process along with the instrumentation arguments. It also provides an easy way for callers to get instrumentation, app context, and an instrumentation arguments `Bundle`. You can get a `UiDevice` object by calling the `getInstance()` method and passing it an `Instrumentation` object—`InstrumentationRegistry.getInstrumentation()`—as the argument. For example:

```
 mDevice = UiDevice
     .getInstance(InstrumentationRegistry.getInstrumentation());
```

4. **Access a UI element displayed on the device:** Get a `UiObject` by calling the `findObject()` method. For example:

```
 UiObject okButton = mDevice.findObject(new UiSelector()
         .text("OK"))
```

```
7.          .className("android.widget.Button"));
```

8. **Perform an action:** Simulate a specific user interaction to perform on that UI element by calling a `UiObject` method. For example:

```
9.  if(okButton.exists() && okButton.isEnabled()) {
10.     okButton.click();
11. }
```

   You may want to call `setText()` to set the text for a `TextView`, or `performMultiPointerGesture()` to simulate a multi-touch gesture.
   You can repeat steps 2 and 3 as needed to test more complex user interactions that involve multiple UI elements or sequences of user actions.

12. **Verify results:** Check that the UI reflects the expected state or behavior after these user interactions are performed. You can use standard JUnit `Assert` methods to test that UI elements in the app return the expected results. For example:

```
13. UiObject result = mDevice.findObject(By.res(CALC_PACKAGE, "result"));
14. assertEquals("5", result.getText());
```

Accessing the device

The `UiDevice` class provides the methods for accessing and manipulating the state of the device. Unlike Espresso, UI Automator can verify the correct behavior of interactions between different user apps, or between user apps and system apps. It lets you interact with visible elements on a device. In your tests, you can call `UiDevice` methods to check for the state of various properties, such as current orientation or display size. Your test can use the `UiDevice` object to perform device-level actions, such as forcing the device into a specific rotation, pressing D-pad hardware buttons, and pressing the Home button.
It's a good practice to start your test from the Home screen of the device. From the Home screen you can call the methods provided by the UI Automator API to select and interact with specific UI elements.

The following code snippet shows how your test can get an instance of `UiDevice`, simulate a Home button press, and launch the app:

```
import org.junit.Before;
import android.support.test.runner.AndroidJUnit4;
import android.support.test.uiautomator.UiDevice;
import android.support.test.uiautomator.By;
import android.support.test.uiautomator.Until;
// ...

@RunWith(AndroidJUnit4.class)
@SdkSuppress(minSdkVersion = 18)
public class ChangeTextBehaviorTest {

    private static final String BASIC_SAMPLE_PACKAGE
            = "com.example.android.testing.uiautomator.BasicSample";
    private static final int LAUNCH_TIMEOUT = 5000;
    private static final String STRING_TO_BE_TYPED = "UiAutomator";
    private UiDevice mDevice;

    @Before
    public void startMainActivityFromHomeScreen() {
        // Initialize UiDevice instance
        mDevice = UiDevice
          .getInstance(InstrumentationRegistry.getInstrumentation());
        // Start from the home screen
        mDevice.pressHome();
        // Wait for launcher
        final String launcherPackage =
                     mDevice.getLauncherPackageName();
        assertThat(launcherPackage, notNullValue());
        mDevice.wait(Until
                .hasObject(By.pkg(launcherPackage).depth(0)),
                LAUNCH_TIMEOUT);
        // Launch the app
        Context context = InstrumentationRegistry.getContext();
        final Intent intent = context.getPackageManager()
                .getLaunchIntentForPackage(BASIC_SAMPLE_PACKAGE);
        // Clear out any previous instances
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
        context.startActivity(intent);
        // Wait for the app to appear
        mDevice.wait(Until
                .hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)),
                LAUNCH_TIMEOUT);
    }
}
```

The `@SdkSuppress(minSdkVersion = 18)` annotation ensures that tests will only run on devices with Android 4.3 (API level 18) or newer, as required by the UI Automator framework.

Accessing a UI element

Use the `findObject()` method of the `UiObject` class to retrieve a `UiObject` instance that represents a UI element matching a given selector criteria. To access a specific UI element, use the `UiSelector` class, which represents a query for specific elements in the currently displayed UI.
You can reuse the `UiObject` instances that you created in other parts of your app testing. The UI Automator test framework searches the current display for a match every time your test uses a UiObject instance to click on a UI element or query an attribute.
The following shows how your test might construct `UiObject` instances using `findObject()` with a `UiSelector` for a **Cancel** button, and one for an **OK** button:

```
UiObject cancelButton = mDevice.findObject(new UiSelector()
        .text("Cancel"))
        .className("android.widget.Button"));
UiObject okButton = mDevice.findObject(new UiSelector()
        .text("OK"))
        .className("android.widget.Button"));
```

If more that one element matches, the first matching element in the layout hierarchy (found by moving from top to bottom, left to right) is returned as the target `UiObject`.
When constructing a `UiSelector`, you can chain together multiple attributes and properties to refine your search. If no matching UI element is found, an exception (`UiAutomatorObjectNotFoundException`) is thrown.
To nest multiple `UiSelector` instances, use the `childSelector()` method of the `UiSelector` class. For example, the following shows how your test might specify a search to find the first `ListView` in the currently displayed UI, then search within that `ListView` to find a UI element with the `android:text` attribute `"List Item 14"`:

```
UiObject appItem = new UiObject(new UiSelector()
        .className("android.widget.ListView")
        .instance(1)
        .childSelector(new UiSelector()
        .text("List Item 14")));
```

While it may be useful to refer to the `android:text` attribute of an element of a `ListView` or `RecycleView` because there is no resource id (`android:id` attribute) for such an element, it is best to use a resource id when specifying a selector rather than the `android:text` or `android:contentDescription` attributes. Not all elements have a text attribute (for example, icons in a toolbar). Tests might fail if there are minor changes to the text of a UI element, and the tests would not be usable for apps translated into other languages because your text selectors would not match the translated string resources.

Performing actions

Once your test has retrieved a `UiObject` object, you can call the methods in the `UiObject` class to perform user interactions on the UI element represented by that object. For example, the constructed `UiObject` instances in the previous section for the **OK** and **Cancel** buttons can be used to perform a click:

```
// Simulate a user-click on the OK button, if found.
if(okButton.exists() && okButton.isEnabled()) {
    okButton.click();
}
```

You can use `UiObject` methods to perform actions such as:

- `click()`: Tap (click) the center of the visible bounds of the UI element.
- `dragTo()`: Drag the object to arbitrary coordinates.
- `setText()`: Set the text in an editable field, after clearing the field's content. Conversely, you use the `clearTextField()` method to clear the existing text in an editable field.
- `swipeUp()`: Perform the swipe up action on the `UiObject`. Similarly, the `swipeDown()`, `swipeLeft()`, and `swipeRight()` methods perform corresponding actions.

Sending an Intent or launching an Activity

The UI Automator testing framework enables you to send an `Intent` or launch an `Activity` without using shell commands, by getting a `Context` object through the `getContext()` method. For example, the following shows how your test can use an `Intent` to launch the app under test:

```
public void setUp() {
    // Setup code ...
    // Launch a simple calculator app.
    Context context = getInstrumentation().getContext();
    Intent intent = context.getPackageManager()
            .getLaunchIntentForPackage(CALC_PACKAGE);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
    // Clear out any previous instances.
    context.startActivity(intent);
    mDevice.wait(Until
                .hasObject(By.pkg(CALC_PACKAGE).depth(0)), TIMEOUT);
}
```

Performing actions on collections

Use the `UiCollection` class if you want to simulate user interactions on a collection of UI elements (for example, song titles or emails in a list). To create a `UiCollection` object, specify a `UiSelector` that searches for a UI container or a wrapper of other child UI elements, such as a layout group that contains child UI elements.

The following shows how a test can use a `UiCollection` to represent a video album that is displayed within a `FrameLayout`:

```
UiCollection videos = new UiCollection(new UiSelector()
        .className("android.widget.FrameLayout"));

// Retrieve the number of videos in this collection:
int count = videos.getChildCount(new UiSelector()
        .className("android.widget.LinearLayout"));

// Find a specific video and simulate a user-click on it
UiObject video = videos.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "Cute Baby Laughing");
video.click();

// Simulate selecting a checkbox that is associated with the video
UiObject checkBox = video.getChild(new UiSelector()
        .className("android.widget.Checkbox"));
if(!checkBox.isSelected()) checkbox.click();
```

Performing actions on scrollable views

Use the `UiScrollable` class to simulate vertical or horizontal scrolling across a display. This technique is helpful when a UI element is positioned off-screen and you need to scroll to bring it into view. For example, the following code snippet shows how to simulate scrolling down the **Settings** menu and clicking on the **About phone** option:

```
UiScrollable settingsItem = new UiScrollable(new UiSelector()
        .className("android.widget.ListView"));
UiObject about = settingsItem.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "About phone");
about.click();
```

Verifying results

You can use standard JUnit Assert methods to test that UI components in the app return the expected results. For example, you can use assertFalse() to assert that a condition is false in order to test if the condition truly is false as a result.
Use assertEquals() to test if a floating point number result is equal to the assertion:

```
assertEquals("5", result.getText());
```

The following shows how your test can locate several buttons in a calculator app, click on them in order, then verify that the correct result is displayed:

```
private static final String CALC_PACKAGE = "com.myexample.calc";
public void testTwoPlusThreeEqualsFive() {
    // Enter an equation: 2 + 3 = ?
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("two")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("plus")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("three")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("equals")).click();

    // Verify the result = 5
    UiObject result = mDevice.findObject(By.res(CALC_PACKAGE, "result"));
    assertEquals("5", result.getText());
}
```

Running instrumented tests

- To run a single test, **right-click** (or **Control-click**) the test in Android Studio, and select **Run** from the pop-up menu.
- To test a method in a test class, **right-click** (or **Control-click**) the method in the test file and click **Run**.
- To run all tests in a directory, **right-click** (or **Control-click**) on the directory and select **Run tests**.

Android Studio displays the results of the test in the Run window.

Related practical

The related practical is 6.1: Espresso for UI testing.

Learn more

Android Studio documentation:

- Test your app

Android developer documentation:

- Test apps on Android
- Fundamentals of Testing
- Testing UI for a single app—Espresso
- Testing UI for multiple apps—UI Automator
- Build instrumented unit tests
- The Hamcrest Tutorial
- Hamcrest API and Utility Classes
- Test support APIs

Android Testing Support Library:

- Espresso documentation
- Espresso samples
- Espresso basics
- Espresso cheat sheet

Videos:

- Android Testing Support - Android Testing Patterns #1 (introduction)
- Android Testing Support - Android Testing Patterns #2 (onView view matching)
- Android Testing Support - Android Testing Patterns #3 (onData and adapter views)

Other:

- Google Testing Blog: Android UI Automated Testing
- Google Testing Blog: Test Sizes
- Atomic Object: "Espresso – Testing RecyclerViews at Specific Positions"
- Stack Overflow: "How to assert inside a RecyclerView in Espresso?"
- GitHub: Android Testing Samples
- Google Codelabs: Android Testing Codelab
- JUnit web site
- JUnit annotations: Package org.junit

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/