UI Testing

In addition to unit testing the individual components that make up your Android application (such as activities, services, and content providers), it is also important that you test the behavior of your application's user interface (UI) when it is running on a device. UI testing ensures that your application returns the correct UI output in response to a sequence of user actions on a device, such as entering keyboard input or pressing toolbars, menus, dialogs, images, and other UI controls.

Functional or black-box UI testing does not require testers to know the internal implementation details of the app, only its expected output when a user performs a specific action or enters a specific input. This approach allows for better separation of development and testing roles in your organization.

One common approach to UI testing is to run tests manually and verify that the app is behaving as expected. However, this approach can be time-consuming, tedious, and error-prone. A more efficient and reliable approach is to automate the UI testing with a software testing framework. Automated testing involves creating programs to perform testing tasks (test cases) to cover specific usage scenarios, and then using the testing framework to run the test cases automatically and in a repeatable manner.

IN THIS DOCUMENT

KEY CLASSES

- IAutomationSupport
- UiAutomatorTestCase
- UiCollection
- UiDevice
- UiObject
- UiScrollable
- UiSelector

SEE ALSO

- uiautomator (reference)

Overview

The Android SDK provides the following tools to support automated, functional UI testing on your application:

- `uiautomatorviewer` - A GUI tool to scan and analyze the UI components of an Android application.
- `uiautomator` - A Java library containing APIs to create customized functional UI tests, and an execution engine to automate and run the tests.

To use these tools, you must have the following versions of the Android development tools installed:

- Android SDK Tools, Revision 21 or higher
- Android SDK Platform, API 16 or higher

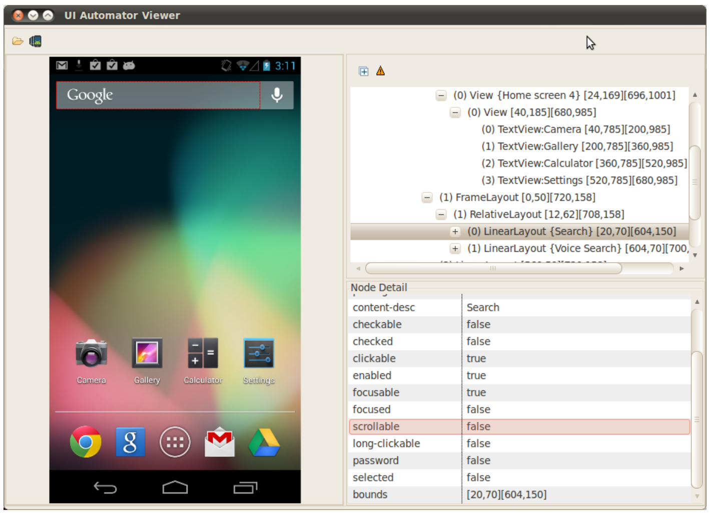
Workflow for the the uiautomator testing framework

Here's a short overview of the steps required to automate UI testing:

1. Prepare to test by installing the app on a test device, analyzing the app's UI components, and ensuring that your application is accessible by the test automation framework.
2. Create automated tests to simulate specific user interactions on your application.
3. Compile your test cases into a JAR file and install it on your test device along with your app.
4. Run the tests and view the test results.
5. Correct any bugs or defects discovered in testing.

Analyzing Your Application's UI

Before you start writing your test cases, it's helpful to familiarize yourself with the UI components (including the views and controls) of the targeted application. You can use the `uiautomatorviewer` tool to take a snapshot of the foreground UI screen on any Android device that is connected to your development machine. The `uiautomatorviewer` tool provides a convenient visual interface to inspect the layout hierarchy and view the properties of the individual UI components that are displayed on the test device. Using this information, you can later create `uiautomator` tests with selector objects that target specific UI components to test.



```
(0) View {Home screen 4} [24,169][696,1001]
  (0) View [40,185][680,985]
    (0) TextView:Camera [40,785][200,985]
    (1) TextView:Gallery [200,785][360,985]
    (2) TextView:Calculator [360,785][520,985]
    (3) TextView:Settings [520,785][680,985]
(1) FrameLayout [0,50][720,158]
  (1) RelativeLayout [12,62][708,158]
    (0) LinearLayout {Search} [20,70][604,150]
    (1) LinearLayout {Voice Search} [604,70][700,
```

Node Detail

content-desc	Search
checkable	false
checked	false
clickable	true
enabled	true
focusable	true
focused	false
scrollable	false
long-clickable	false
password	false
selected	false
bounds	[20,70][604,150]

Figure 1. The `uiautomatorviewer` showing the captured interface of a test deviice.

To analyze the UI components of the application that you want to test:

1. Connect your Android device to your development machine.
2. Open a terminal window and navigate to `<android-sdk>/tools/`.
3. Run the tool with this command:

   ```
   $ uiautomatorviewer
   ```

4. To capture a screen for analysis, click the **Device Screenshot** button in the GUI of the `uiautomatorviewer` tool.

   > **Note:** If you have more than one device connected, specify the device for screen capture by setting the `ANDROID_SERIAL` environment variable:

   a. Find the serial numbers for your connected devices by running this command:

      ```
      $ adb devices
      ```

   b. Set the `ANDROID_SERIAL` environment variable to select the device to test:
   - In Windows:

      ```
      set ANDROID_SERIAL=<device serial number>
      ```

   - In UNIX:

      ```
      export ANDROID_SERIAL=<device serial number>
      ```

   If you are connected to only a single device, you do not need to set the ANDROID_SERIAL environment variable.

5. View the UI properties for your application:
   - Hover over the snapshot in the left-hand panel to see the UI components identified by the `uiautomatorviewer` tool. You can view the component's properties listed in the lower right-hand panel, and the layout hierarchy in the upper right-hand panel.
   - Optionally, click on the **Toggle NAF Nodes** button to see UI components that are not accessible to the `uiautomator` testing framework. Only limited information may be available for these components.

Preparing to Test

Before using the `uiautomator` testing framework, complete these pre-flight tasks:

Load the application to a device

If you are reading this document, chances are that the Android application that you want to test has not been published yet. If you have a copy of the APK file, you can install the APK onto a test device by using the `adb` tool. To learn how to install an APK file using the `adb` tool, see the adb (/tools/help/adb.html#move) documentation.

Identify the application's UI components

Before writing your `uiautomator` tests, first identify the UI components in the application that you want to test. Typically, good candidates for testing are UI components that are visible and that users can interact with. The UI components should also have visible text labels, `android:contentDescription` (/reference/android/view/View.html#attr_android:contentDescription) values, or both.

You can inspect the visible screen objects in an application conveniently by using the `uiautomatorviewer` tool. For more information about how to analyze an application screen with this tool, see the section Analyzing Your Application's UI (#uianalaysis). For more information about the common types of UI components provided by Android, see User Interface (/guide/topics/ui/index.html).

Ensure that the application is accessible

This step is required because the `uiautomator` tool depends on the accessibility features of the Android framework to execute your functional UI tests. You should include these minimum optimizations to support the `uiautomator` tool:

- Use the `android:contentDescription` attribute to label the `ImageButton`, `ImageView`, `CheckBox` and other user interface controls.
- Provide an `android:hint` attribute *instead* of a content description for `EditText` fields
- Associate an `android:hint` attribute with any graphical icons used by controls that provide feedback to the user (for example, status or state information).
- Make sure that all the user interface elements are accessible with a directional controller, such as a trackball or D-pad.
- Use the `uiautomatorviewer` tool to ensure that the UI component is accessible to the testing framework. You can also test the application by turning on accessibility services like TalkBack and Explore by Touch, and try using your application using only directional controls.

For more information about implementing and testing accessibility, see Making Applications Accessible (/guide/topics/ui/accessibility/apps.html).

> **Note:** To identify the non-accessible components in the UI, click on the **Toggle NAF Nodes** option in the `uiautomatorviewer` tool.

Generally, Android application developers get accessibility support for free, courtesy of the `View` `(/reference/android/view/View.html)` and `ViewGroup` `(/reference/android/view/ViewGroup.html)` classes. However, some applications use custom view components to provide a richer user experience. Such custom components won't get the accessibility support that is provided by the standard Android UI components. If this applies to your application, ensure that the application developer exposes the custom drawn UI components to Android accessibility services, by implementing the `AccessibilityNodeProvider` `(/reference/android/view/accessibility/AccessibilityNodeProvider.html)` class. For more information about making custom view components accessible, see Making Applications Accessible (/guide/topics/ui/accessibility/apps.html#custom-views).

Configure your development environment

If you're developing in Eclipse, the Android SDK provides additional tools that help you write test cases using `uiautomator` and buiild your JAR file. In order to set up Eclipse to assist you, you need to create a project that includes the `uiautomator` client library, along with the Android SDK library. To configure Eclipse:

1. Create a new Java project in Eclipse, and give your project a name that is relevant to the tests you're about to create (for example, "MyAppNameTests"). In the project, you will create the test cases that are specific to the application that you want to test.
2. From the **Project Explorer**, right-click on the new project that you created, then select **Properties > Java Build Path**, and do the following:
   a. Click **Add Library > JUnit** then select **JUnit3** to add JUnit support.
   b. Click **Add External JARs**... and navigate to the SDK directory. Under the platforms directory, select the latest SDK version and add both the `uiautomator.jar` and `android.jar` files.

If you did not configure Eclipse as your development environment, make sure that the `uiautomator.jar` and `android.jar` files from the `<android-sdk>/platforms/<sdk>` directory are in your Java class path.

Once you have completed these prerequisite tasks, you're almost ready to start creating your `uiautomator` tests.

Creating uiautomator Tests

To build a test that runs in the `uiautomator` framework, create a test case that extends the `UiAutomatorTestCase` (/tools/help/uiautomator/UiAutomatorTestCase.html) class. In Eclipse, the test case file goes under the `src` directory in your project.

Later, you will build the test case as a JAR file, then copy this file to the test device. The test JAR file is not an APK file and resides separately from the application that you want to test on the device.

Because the `UiAutomatorTestCase` (/tools/help/uiautomator/UiAutomatorTestCase.html) class extends `junit.framework.TestCase`, you can use the JUnit `Assert` class to test that UI components in the app return the expected results. To learn more about JUnit, you can read the documentation on the junit.org (http://www.junit.org/) home page.

The first thing your test case should do is access the device that contains the target app. It's also good practice to start the test from the Home screen of the device. From the Home screen (or some other starting location you've chosen in the target app), you can use the classes provided by the `uiautomator` API to simulate user actions and to test specific UI components. For an example of how to put together a `uiautomator` test case, see the sample test case (#sample).

uiautomator API

The `uiautomator` API is bundled in the `uiautomator.jar` file under the `<android-sdk>/platforms/` directory. The API includes these key classes that allow you to capture and manipulate UI components on the target app:

`UiDevice`

Represents the device state. In your tests, you can call methods on the `UiDevice` (/tools/help/uiautomator/UiDevice.html) instance to check for the state of various properties, such as current orientation or display size. Your tests also can use the `UiDevice` (/tools/help/uiautomator/UiDevice.html) instance to perform device level actions, such as forcing the device into a specific rotation, pressing the d-pad hardware button, or pressing the Home and Menu buttons.

To get an instance of `UiDevice` (/tools/help/uiautomator/UiDevice.html) and simulate a Home button press:

```
getUiDevice().pressHome();
```

`UiSelector`

Represents a search criteria to query and get a handle on specific elements in the currently displayed UI. If more than one matching element is found, the first matching element in the layout hierarchy is returned as the target `UiObject`. When constructing a `UiSelector`, you can chain together multiple properties to refine your search. If no matching UI element is found, a `UiAutomatorObjectNotFoundException` is thrown. You can use the `childSelector()` method to nest multiple `UiSelector` instances. For example, the following code example shows how to 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 text property `Apps`.

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

`UiObject`

Represents a UI element. To create a `UiObject` instance, use a `UiSelector` that describes how to search for, or select, the UI element.

The following code example shows how to construct `UiObject` (/tools/help/uiautomator/UiObject.html) instances that represent a **Cancel** button and a **OK** button in your application.

```
UiObject cancelButton = new UiObject(new UiSelector().text("Cancel"));
UiObject okButton = new UiObject(new UiSelector().text("OK"));
```

You can reuse the `UiObject` (/tools/help/uiautomator/UiObject.html) instances that you have created in other parts of your app testing, as needed. Note that the `uiautomator` test framework searches the current display for a match every time your test uses a `UiObject` (/tools/help/uiautomator/UiObject.html) instance to click on a UI element or query a property.

In the following code example, the `uiautomator` test framework searches for a UI element with the text property `OK`. If a match is found and if the element is enabled, the framework simulates a user click action on the element.

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

You can also restrict the search to find only elements of a specific class. For example, to find matches of the `Button` (/reference/android/widget/Button.html) class:

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

`UiCollection`

Represents a collection of items, for example songs in a music album or a list of emails in an inbox. Similar to a `UiObject`, you construct a `UiCollection` instance by specifying a `UiSelector`. The `UiSelector` for a `UiCollection` should search for a UI element that is a container or wrapper of other child UI elements (such as a layout view that contains child UI elements). For example, the following code snippet shows how to construct a `UiCollection` to represent a video album that is displayed within a `FrameLayout`:

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

If the videos are listed within a `LinearLayout` (/reference/android/widget/LinearLayout.html) view, and you want to to retrieve the number of videos in this collection:

```
int count = videos.getChildCount(new UiSelector()
    .className("android.widget.LinearLayout"));
```

If you want to find a specific video that is labeled with the text element `Cute Baby Laughing` from the collection and simulate a user-click on the video:

```
UiObject video = videos.getChildByText(new UiSelector()
    .className("android.widget.LinearLayout"), "Cute Baby Laughing");
video.click();
```

Similarly, you can simulate other user actions on the UI object. For example, if you want to 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();
```

`UiScrollable`

Represents a scrollable collection of UI elements. You can 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 shows how to simulate scrolling down the Settings menu and clicking on an **About tablet** option:

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

For more information about these APIs, see the `uiautomator` (/tools/help/uiautomator/index.html) reference.

A sample uiautomator test case

The following code example shows a simple test case which simulates a user bringing up the Settings app in a stock Android device. The test case mimics all the steps that a user would typically take to perform this task, including opening the Home screen, launching the **All Apps** screen, scrolling to the **Settings** app icon, and clicking on the icon to enter the Settings app.

```java
package com.uia.example.my;

// Import the uiautomator libraries
import com.android.uiautomator.core.UiObject;
import com.android.uiautomator.core.UiObjectNotFoundException;
import com.android.uiautomator.core.UiScrollable;
import com.android.uiautomator.core.UiSelector;
import com.android.uiautomator.testrunner.UiAutomatorTestCase;

public class LaunchSettings extends UiAutomatorTestCase {

   public void testDemo() throws UiObjectNotFoundException {

      // Simulate a short press on the HOME button.
      getUiDevice().pressHome();

      // We're now in the home screen. Next, we want to simulate
      // a user bringing up the All Apps screen.
      // If you use the uiautomatorviewer tool to capture a snapshot
      // of the Home screen, notice that the All Apps button's
      // content-description property has the value "Apps".  We can
      // use this property to create a UiSelector to find the button.
      UiObject allAppsButton = new UiObject(new UiSelector()
         .description("Apps"));

      // Simulate a click to bring up the All Apps screen.
      allAppsButton.clickAndWaitForNewWindow();

      // In the All Apps screen, the Settings app is located in
      // the Apps tab. To simulate the user bringing up the Apps tab,
      // we create a UiSelector to find a tab with the text
      // label "Apps".
      UiObject appsTab = new UiObject(new UiSelector()
         .text("Apps"));

      // Simulate a click to enter the Apps tab.
      appsTab.click();

      // Next, in the apps tabs, we can simulate a user swiping until
      // they come to the Settings app icon.  Since the container view
      // is scrollable, we can use a UiScrollable object.
      UiScrollable appViews = new UiScrollable(new UiSelector()
         .scrollable(true));

      // Set the swiping mode to horizontal (the default is vertical)
      appViews.setAsHorizontalList();

      // Create a UiSelector to find the Settings app and simulate
      // a user click to launch the app.
      UiObject settingsApp = appViews.getChildByText(new UiSelector()
         .className(android.widget.TextView.class.getName()),
         "Settings");
      settingsApp.clickAndWaitForNewWindow();

      // Validate that the package name is the expected one
      UiObject settingsValidation = new UiObject(new UiSelector()
         .packageName("com.android.settings"));
      assertTrue("Unable to detect Settings",
         settingsValidation.exists());
   }
}
```

Building and Deploying Your uiautomator Tests

Once you have coded your test, follow these steps to build and deploy your test JAR to your target Android test device:

1. Create the required build configuration files to build the output JAR. To generate the build configuration files, open a terminal and run the following command:

   ```
   <android-sdk>/tools/android create uitest-project -n <name> -t 1 -p <path>
   ```

   The `<name>` is the name of the project that contains your `uiautomator` test source files, and the `<path>` is the path to the corresponding project directory.
2. From the command line, set the `ANDROID_HOME` variable:
   - In Windows:

     ```
     set ANDROID_HOME=<path_to_your_sdk>
     ```

   - In UNIX:

     ```
     export ANDROID_HOME=<path_to_your_sdk>
     ```

3. Go to the project directory where your `build.xml` file is located and build your test JAR.

   ```
   ant build
   ```

4. Deploy your generated test JAR file to the test device by using the `adb push` command:

   ```
   adb push <path_to_output_jar> /data/local/tmp/
   ```

   Here's an example:

   ```
   adb push ~/dev/workspace/LaunchSettings/bin/LaunchSettings.jar /data/local/tmp/
   ```

Running uiautomator Tests

Here's an example of how to run a test that is implemented in the `LaunchSettings.jar` file. The tests are bundled in the `com.uia.example.my` package:

```
adb shell uiautomator runtest LaunchSettings.jar -c com.uia.example.my.LaunchSettings
```

To learn more about the syntax, subcommands, and options for `uiautomator`, see the `uiautomator` (/tools/help/uiautomator/index.html) reference.

Best Practices

Here are some best practices for functional UI testing with the `uiautomator` framework:

- Ensure that you validate the same UI functions on your application across the various types of devices that your application might run on (for example, devices with different screen densities).
- You should also test your UI against common scenarios such as in-coming phone calls, network interruptions, and user-initiated switching to other applications on the device.