|  |  |  |
| --- | --- | --- |
|  | **2, LEBUH ACHEH, GEORGE TOWN**  **10300 GEORGE TOWN**  **PULAU PINANG**  **INFORMATION SHEET** | |
| **PROGRAM’S CODE & NAME** | J620-002-4:2020 FRONT-END SOFTWARE DEVELOPMENT | |
| **LEVEL** | FOUR (4) | |
| **COMPETENCY UNIT NO. AND TITLE** | J620-002-4:2020-C04 MOBILE APPLICATION WITH THIRD PARTY API DEVELOPMENT | |
| **WORK ACTIVITIES NO. AND STATEMENT** | 1. CREATE MOBILE APP DESIGN MOCK-UP ELEMENTS. 2. PLAN MOBILE APP DESIGN STRUCTURE. 3. TRANSFORM MOCK-UP TO MOBILE APP. 4. INTEGRATE MOBILE APP WITH DATA SOURCE. 5. **VERIFY SUCCESSFUL API INTEGRATION** 6. VERIFY DEVELOPED MOBILE APP. 7. VERIFY MOBILE APP ACCESSIBLE GLOBALLY. | |
| **CODE NO.** | J620-002-4:2020-C04/IS(10/15) | Page: 1 of39 |

**TITLE**:

**USER ACCEPTANCE TEST**

**PURPOSE**:

This information sheet is intended to provide insight and knowledge to trainees with regards to the fundamentals of testing the application.

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of testing the application.

# **FUNDAMENTALS OF TESTING**

Users interact with your app on a variety of levels, from pressing a button to downloading information onto their device. Accordingly, you should test a variety of use cases and interactions as you iteratively develop your app.

## Organizing your code for testing

As your app expands, you might find it necessary to fetch data from a server, interact with the device's sensors, access local storage, or render complex user interfaces. The versatility of your app demands a comprehensive testing strategy.

1. Create and test code iteratively

When developing a feature iteratively, you start by either writing a new test or by adding cases and assertions to an existing unit test. The test fails at first because the feature isn't implemented yet.

It's important to consider the units of responsibility that emerge as you design the new feature. For each unit, you write a corresponding unit test. Your unit tests should nearly exhaust all possible interactions with the unit, including standard interactions, invalid inputs, and cases where resources aren't available. Take advantage of Jetpack libraries whenever possible; when you use these well-tested libraries, you can focus on validating behavior that's specific to your app.

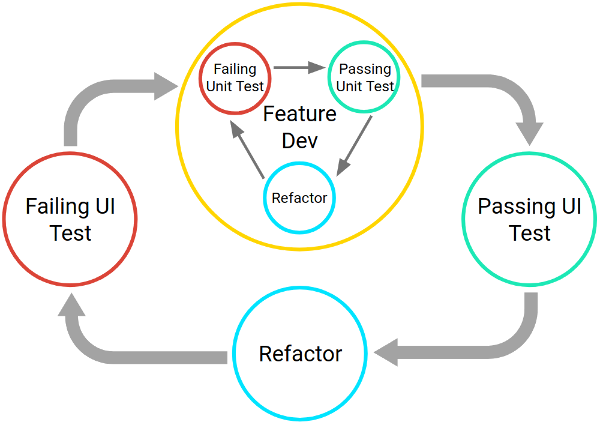


Figure 1: Test-Driven Development

1. View your app as a series of modules

To make your code easier to test, develop your code in terms of modules, where each module represents a specific task that users complete within your app. This perspective contrasts the stack-based view of an app that typically contains layers representing the UI, business logic, and data.

For example, a "task list" app might have modules for creating tasks, viewing statistics about completed tasks, and taking photographs to associate with a particular task. Such a modular architecture also helps you keep unrelated classes decoupled and provides a natural structure for assigning ownership within your development team.

It's important to set well-defined boundaries around each module, and to create new modules as your app grows in scale and complexity. Each module should have only one area of focus, and the APIs that allow for inter-module communication should be consistent. To make it easier and quicker to test these inter-module interactions, consider creating fake implementations of your modules. In your tests, the real implementation of one module can call the fake implementation of the other module.

As you create a new module, however, don't be too dogmatic about making it full-featured right away. It's OK for a particular module to not have one or more layers of the app's stack.

To learn more about how to define modules in your app, as well as platform support for creating and publishing modules, see Android App Bundles.

## Configuring your test environment

When setting up your environment and dependencies for creating tests in your app, follow the best practices described in this section.

1. Organize test directories based on execution environment

A typical project in Android Studio contains two directories in which you place tests. Organize your tests as follows:

* The androidTest directory should contain the tests that run on real or virtual devices. Such tests include integration tests, end-to-end tests, and other tests where the JVM alone cannot validate your app's functionality.
* The test directory should contain the tests that run on your local machine, such as unit tests.

1. Consider trade offs of running tests on different types of devices

When running your tests on a device, you can choose among the following types:

1. Real device
2. Virtual device (such as the emulator in Android Studio)
3. Simulated device (such as Robolectric)

Real devices offer the highest fidelity but also take the most time to run your tests. Simulated devices, on the other hand, provide improved test speed at the cost of lower fidelity. The platform's improvements in binary resources and realistic loopers, however, allow simulated devices to produce more realistic results.

Virtual devices offer a balance of fidelity and speed. When you use virtual devices to test, use snapshots to minimize setup time in between tests.

1. Consider whether to use test doubles

When creating tests, you have the option of creating real objects or test doubles, such as fake objects or mock objects. Generally, using real objects in your tests is better than using test doubles, especially when the object under test satisfies one of the following conditions:

1. The object is a data object.
2. The object cannot function unless it communicates with the real object version of a dependency. A good example is an event callback handler.
3. It's hard to replicate the object's communication with a dependency. A good example is a SQL database handler, where an in-memory database provides more robust tests than fakes of database results.

In particular, mocking instances of types that you don't own usually leads to brittle tests that work only when you've understood the complexities of someone else's implementation of that type. Use such mocks only as a last resort. It's OK to mock your own objects, but keep in mind that mocks annotated using @Spy provide more fidelity than mocks that stub out all functionality within a class.

However, it's better to create fake or even mock objects if your tests try to perform the following types of operations on a real object:

1. Long operations, such as processing a large file.
2. Non-hermetic actions, such as connecting to an arbitrary open port.
3. Hard-to-create configurations.

## Writing your tests

After you've configured your testing environment, it's time to write tests that evaluate your app's functionality. This section describes how to write small, medium, and large tests.

1. Levels of the Testing Pyramid

The Testing Pyramid, shown in Figure 2, illustrates how your app should include the three categories of tests: small, medium, and large:

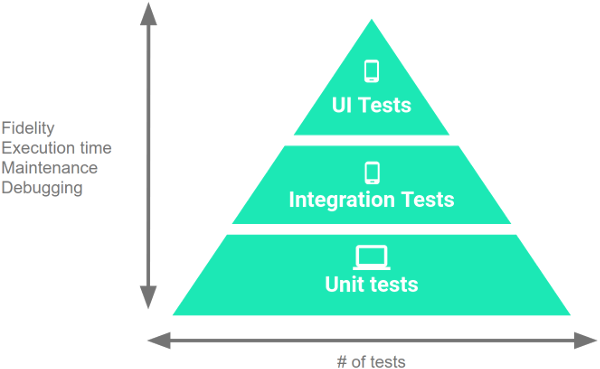


Figure 2: The Testing Pyramid

1. Small tests are unit tests that validate your app's behavior one class at a time.
2. Medium tests are integration tests that validate either interactions between levels of the stack within a module, or interactions between related modules.
3. Large tests are end-to-end tests that validate user journeys spanning multiple modules of your app.

As you work up the pyramid, from small tests to large tests, each test increases in fidelity but also increases in execution time and effort to maintain and debug. Therefore, you should write more unit tests than integration tests, and more integration tests than end-to-end tests. Although the proportion of tests for each category can vary based on your app's use cases, we generally recommend the following split among the categories: 70 percent small, 20 percent medium, and 10 percent large.

To learn more about the Android Testing Pyramid, see the Test-Driven Development on Android session video from Google I/O 2017, starting at 1:51.

1. Write small tests

The small tests that you write should be highly-focused unit tests that exhaustively validate the functionality and contracts of each class within your app.

As you add and change methods within a particular class, create and run unit tests against them. If these tests depend on the Android framework, use a unified, device-agnostic API, such as the androidx.test APIs. This consistency allows you to run your test locally without a physical device or emulator.

If your tests rely on resources, enable the includeAndroidResources option in your app's build.gradle file. Your unit tests can then access compiled versions of your resources, allowing the tests to run more quickly and accurately.

app/build.gradle

android {

// ...

testOptions.unitTests.includeAndroidResources = true

}

1. Local unit tests

Use the AndroidX Test APIs whenever possible so that your unit tests can run on a device or emulator. For tests that always run on a JVM-powered development machine, you can use Robolectric.

Robolectric simulates the runtime for Android 4.1 (API level 16) or higher and provides community-maintained fakes called shadows. This functionality allows you to test code that depends on the framework without needing to use an emulator or mock objects. Robolectric supports the following aspects of the Android platform:

1. Component lifecycles
2. Event loops
3. All resources
4. Instrumented unit tests

You can run instrumented unit tests on a physical device or emulator. This form of testing involves significantly slower execution times than local unit tests, however, so it's best to rely on this method only when it's essential to evaluate your app's behavior against actual device hardware.

When running instrumented tests, AndroidX Test makes use of the following threads:

1. The main thread, also known as the "UI thread" or the "activity thread", where UI interactions and activity lifecycle events occur.
2. The instrumentation thread, where most of your tests run. When your test suite begins, the AndroidJUnitTest class starts this thread.

If you need a test to execute on the main thread, annotate it using @UiThreadTest.

1. Write medium tests

In addition to testing each unit of your app by running small tests, you should validate your app's behavior from the module level. To do so, write medium tests, which are integration tests that validate the collaboration and interaction of a group of units.

Use your app's structure and the following examples of medium tests (in order of increasing scope) to define the best way to represent groups of units in your app:

1. Interactions between a view and view model, such as testing a Fragment object, validating layout XML, or evaluating the data-binding logic of a ViewModel object.
2. Tests in your app's repository layer, which verify that your different data sources and data access objects (DAOs) interact as expected.
3. Vertical slices of your app, testing interactions on a particular screen. Such a test verifies the interactions throughout the layers of your app's stack.
4. Multi-fragment tests that evaluate a specific area of your app. Unlike the other types of medium tests mentioned in this list, this type of test usually requires a real device because the interaction under test involves multiple UI elements.

To carry out these tests, do the following:

1. Use methods from the Espresso Intents library. To simplify the information that you're passing into these tests, use fakes and stubbing.
2. Combine your use of IntentSubject and Truth-based assertions to verify the captured intents.
3. Use Espresso when running instrumented medium tests

Espresso helps keep tasks synchronized as you perform UI interactions similar to the following on a device or on Robolectric:

1. Performing actions on View objects.
2. Assessing how users with accessibility needs can use your app.
3. Locating and activating items within RecyclerView and AdapterView objects.
4. Validating the state of outgoing intents.
5. Verifying the structure of a DOM within WebView objects.

To learn more about these interactions and how to use them in your app's tests, see the Espresso guide.

1. Write large tests

Although it's important to test each class and module within your app in isolation, it's just as important to validate end-to-end workflows that guide users through multiple modules and features. These types of tests form unavoidable bottlenecks in your code, but you can minimize this effect by validating an app that's as close to the actual, finished product as possible.

If your app is small enough, you might need only one suite of large tests to evaluate your app's functionality as a whole. Otherwise, you should divide your large test suites by team ownership, functional verticals, or user goals.

Typically, it's better to test your app on an emulated device or a cloud-based service like Firebase Test Lab, rather than on a physical device, as you can test multiple combinations of screen sizes and hardware configurations more easily and quickly.

1. Synchronization support in Espresso

In addition to supporting medium-sized instrumentation tests, Espresso provides support for synchronization when completing the following tasks in large tests:

1. Completing workflows that cross your app's process boundaries. Available only on Android 8.0 (API level 26) and higher.
2. Tracking long-running background operations within your app.
3. Performing off-device tests.

To learn more about these interactions and how to use them in your app's tests, see the Espresso guide.

1. Complete other testing tasks using AndroidX Test

This section describes how to use elements of AndroidX Test to further refine your app's tests.

1. Create more readable assertions using Truth

The Guava team provides a fluent assertions library called Truth. You can use this library as an alternative to JUnit- or Hamcrest-based assertions when constructing the validation step—or then step—of your tests.

Usually, you use Truth to express that a particular object has a specific property using phrases that contain the conditions you're testing, such as the following:

1. assertThat(object).hasFlags(FLAGS)
2. assertThat(object).doesNotHaveFlags(FLAGS)
3. assertThat(intent).hasData(URI)
4. assertThat(extras).string(string\_key).equals(EXPECTED)

AndroidX Test supports several additional subjects for Android to make Truth-based assertions even easier to construct:

1. BundleSubject
2. IntentSubject
3. MotionEventSubject
4. NotificationActionSubject
5. NotificationSubject
6. PendingIntentSubject
7. PointerCoordsSubject
8. PointerPropertiesSubject

The AndroidX Test API helps you carry out common tasks related to mobile app testing, which the following sections discuss.

1. Write UI tests

Espresso allows you to programmatically locate and interact with UI elements in your app in a thread-safe way. To learn more, see the Espresso guide.

1. Run UI tests

The AndroidJUnitRunner class defines an instrumentation-based JUnit test runner that lets you run JUnit 3- or JUnit 4-style test classes on Android devices. The test runner facilitates loading your test package and the app under test onto a device or emulator, running your tests, and reporting the results.

To further increase these tests' reliability, use Android Test Orchestrator, which runs each UI test in its own Instrumentation sandbox. This architecture reduces shared state between tests and isolates app crashes on a per-test basis. For more information about the benefits that Android Test Orchestrator provides as you test your app, see the Android Test Orchestrator guide.

1. Interact with visible elements

The UI Automator API lets you interact with visible elements on a device, regardless of the activity or fragment that has focus.

1. Add accessibility checks to validate general usability

Your app's interface should allow all users, including those with accessibility needs, to interact with the device and complete tasks more easily in your app.

To help validate your app's accessibility, Android's testing library provides several pieces of built-in functionality, which is discussed in the following sections. To learn more about how to validate your app's usability for different types of users, see the guide on testing your app's accessibility.

1. Robolectric

Enable accessibility checks by including the @AccessibilityChecks annotation at the beginning of your test suite, as shown in the following code snippet:

import org.robolectric.annotation.AccessibilityChecks

@AccessibilityChecks

class MyTestSuite {

// Your tests here.

}

1. Espresso

Enable accessibility checks by calling AccessibilityChecks.enable() in your test suite's setUp() method, as shown in the following code snippet.

For more information on how to interpret the results of these accessibility checks, see the Espresso accessibility checking guide.

import androidx.test.espresso.accessibility.AccessibilityChecks

@Before

fun setUp() {

AccessibilityChecks.enable()

}

1. Drive activity and fragment lifecycles

Use the ActivityScenario and FragmentScenario classes to test how your app's activities and fragments respond to system-level interruptions and configuration changes. To learn more, see the guides on how to test activities and test fragments.

1. Manage service lifecycles

AndroidX Test includes code for managing the lifecycles of key services. To learn how to define these rules, see the JUnit4 Rules guide.

1. Evaluate all variants of behavior that differ by SDK version

If your app's behavior depends on the device's SDK version, use the @SdkSuppress annotation, passing in values for minSdkVersion or maxSdkVersion depending on how you've branched your app's logic:

@Test

@SdkSuppress(maxSdkVersion = 27)

fun testButtonClickOnOreoAndLower() {

// ...

}

@Test

@SdkSuppress(minSdkVersion = 28)

fun testButtonClickOnPieAndHigher() {

// ...

}

# **AUTOMATING USER INTERFACE TESTS**

User interface (UI) testing lets you ensure that your app meets its functional requirements and achieves a high standard of quality such that it is more likely to be successfully adopted by users.

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. A more efficient approach is to write your UI tests such that user actions are performed in an automated way. The automated approach allows you to run your tests quickly and reliably in a repeatable manner.

To automate UI tests with Android Studio, you implement your test code in a separate Android test folder (src/androidTest/java). The Android Plug-in for Gradle builds a test app based on your test code, then loads the test app on the same device as the target app. In your test code, you can use UI testing frameworks to simulate user interactions on the target app, in order to perform testing tasks that cover specific usage scenarios.

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

* UI tests that span a single app: This type of test verifies that the target app behaves as expected when a user performs a specific action or enters a specific input in its activities. It allows you to check that the target app returns the correct UI output in response to user interactions in the app’s activities. 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: This type of test verifies the correct behavior of interactions between different user apps or between user apps and system apps. For example, you might want to test that your camera app shares images correctly with a 3rd-party social media app, or with the default Android Photos app. UI testing frameworks that support cross-app interactions, such as UI Automator, allow you to create tests for such scenarios.

The lessons in this class teach you how to use the tools and APIs in AndroidX Test to build these types of automated tests. Before you begin building tests using these APIs, add AndroidX Test, as described in Set up project for AndroidX Test.

## Testing UI for a single app

Testing user interactions within a single app helps to ensure that users do not encounter unexpected results or have a poor experience when interacting with your app. You should get into the habit of creating user interface (UI) tests if you need to verify that the UI of your app is functioning correctly.

The Espresso testing framework, provided by AndroidX Test, provides APIs for writing UI tests to simulate user interactions within a single target app. Espresso tests can run on devices running Android 4.0.1 (API level 14) and higher. A key benefit of using Espresso is that it provides automatic synchronization of test actions with the UI of the app you are testing. Espresso detects when the main thread is idle, so it is able to run your test commands at the appropriate time, improving the reliability of your tests. This capability also relieves you from having to add any timing workarounds, such as Thread.sleep() in your test code.

The Espresso testing framework is an instrumentation-based API and works with the AndroidJUnitRunner test runner.

Set up Espresso

Before building your UI test with Espresso, make sure to set a dependency reference to the Espresso library:

1. Set up Espresso

Before building your UI test with Espresso, make sure to set a dependency reference to the Espresso library:

dependencies {

androidTestImplementation('androidx.test.espresso:espresso-core:3.1.0')

}

Turn off animations on your test device — leaving system animations turned on in the test device might cause unexpected results or may lead your test to fail. Turn off animations from Settings by opening Developer options and turning all the following options off:

1. Window animation scale
2. Transition animation scale
3. Animator duration scale

If you want to set up your project to use Espresso features other than what the core API provides, see the guides specific to Espresso.

1. Create an Espresso test class

To create an Espresso test, follow this programming model:

1. Find the UI component you want to test in an Activity (for example, a sign-in button in the app) by calling the onView() method, or the onData() method for AdapterView controls
2. Simulate a specific user interaction to perform on that UI component, by calling the ViewInteraction.perform() or DataInteraction.perform() method and passing in the user action (for example, click on the sign-in button). To sequence multiple actions on the same UI component, chain them using a comma-separated list in your method argument.
3. Repeat the steps above as necessary, to simulate a user flow across multiple activities in the target app.
4. Use the ViewAssertions methods to check that the UI reflects the expected state or behavior, after these user interactions are performed.

These steps are covered in more detail in the sections below.

The following code snippet shows how your test class might invoke this basic workflow:

onView(withId(R.id.my\_view)) // withId(R.id.my\_view) is a ViewMatcher

.perform(click()) // click() is a ViewAction

.check(matches(isDisplayed())) // matches(isDisplayed()) is aViewAssertion

1. Use Espresso with ActivityScenarioRule

The following section describes how to create a new Espresso test in the JUnit 4 style and use ActivityScenarioRule to reduce the amount of boilerplate code you need to write. By using ActivityScenarioRule, the testing framework launches the activity under test before each test method annotated with @Test and before any method annotated with @Before. The framework handles shutting down the activity after the test finishes and all methods annotated with @After are run.

package com.example.android.testing.espresso.BasicSample

import org.junit.Before

import org.junit.Rule

import org.junit.Test

import org.junit.runner.RunWith

import androidx.test.ext.junit.rules.ActivityScenarioRule

import androidx.test.ext.junit.runners.AndroidJUnit4

@RunWith(AndroidJUnit4::class)

@LargeTest

class ChangeTextBehaviorTest {

private lateinit var stringToBetyped: String

@get:Rule

var activityRule: ActivityScenarioRule<MainActivity>

= ActivityScenarioRule(MainActivity::class.java)

@Before

fun initValidString() {

// Specify a valid string.

stringToBetyped = "Espresso"

}

@Test

fun changeText\_sameActivity() {

// Type text and then press the button.

onView(withId(R.id.editTextUserInput))

.perform(typeText(stringToBetyped), closeSoftKeyboard())

onView(withId(R.id.changeTextBt)).perform(click())

// Check that the text was changed.

onView(withId(R.id.textToBeChanged))

.check(matches(withText(stringToBetyped)))

}

}

1. Access UI components

Before Espresso can interact with the app under test, you must first specify the UI component or view. Espresso supports the use of Hamcrest matchers for specifying views and adapters in your app.

To find the view, call the onView() method and pass in a view matcher that specifies the view that you are targeting. This is described in more detail in Specify a view matcher. The onView() method returns a ViewInteraction object that allows your test to interact with the view. However, calling the onView() method may not work if you want to locate a view in an RecyclerView layout. In this case, follow the instructions in Locating a view in an AdapterView instead.

The following code snippet shows how you might write a test that accesses an EditText field, enters a string of text, closes the virtual keyboard, and then performs a button click.

fun testChangeText\_sameActivity() {

// Type text and then press the button.

onView(withId(R.id.editTextUserInput))

.perform(typeText(STRING\_TO\_BE\_TYPED), closeSoftKeyboard())

onView(withId(R.id.changeTextButton)).perform(click())

// Check that the text was changed.

...

}

1. Specify a view matcher

You can specify a view matcher by using these approaches:

1. Calling methods in the ViewMatchers class. For example, to find a view by looking for a text string it displays, you can call a method like this:

onView(withText("Sign-in"))

Similarly you can call withId() and providing the resource ID (R.id) of the view, as shown in the following example:

onView(withId(R.id.button\_signin))

Android resource IDs are not guaranteed to be unique. If your test attempts to match to a resource ID used by more than one view, Espresso throws an AmbiguousViewMatcherException.

1. Using the Hamcrest Matchers class. You can use the allOf() methods to combine multiple matchers, such as containsString() and instanceOf(). This approach allows you to filter the match results more narrowly, as shown in the following example:

onView(allOf(withId(R.id.button\_signin), withText("Sign-in")))

You can use the not keyword to filter for views that don't correspond to the matcher, as shown in the following example:

onView(allOf(withId(R.id.button\_signin), not(withText("Sign-out"))))

To use these methods in your test, import the org.hamcrest.Matchers package. To learn more about Hamcrest matching, see the Hamcrest site.

To improve the performance of your Espresso tests, specify the minimum matching information needed to find your target view. For example, if a view is uniquely identifiable by its descriptive text, you do not need to specify that it is also assignable from the TextView instance.

1. Locate a view in an AdapterView

In an AdapterView widget, the view is dynamically populated with child views at runtime. If the target view you want to test is inside an AdapterView (such as a ListView, GridView, or Spinner), the onView() method might not work because only a subset of the views may be loaded in the current view hierarchy.

Instead, call the onData() method to obtain a DataInteraction object to access the target view element. Espresso handles loading the target view element into the current view hierarchy. Espresso also takes care of scrolling to the target element and putting the element into focus.

The following code snippet shows how you can use the onData() method together with Hamcrest matching to search for a specific row in a list that contains a given string. In this example, the LongListActivity class contains a list of strings exposed through a SimpleAdapter.

onData(allOf(`is`(instanceOf(Map::class.java)),

hasEntry(equalTo(LongListActivity.ROW\_TEXT),

`is`("test input"))))

1. Perform actions

Call the ViewInteraction.perform() or DataInteraction.perform() methods to simulate user interactions on the UI component. You must pass in one or more ViewAction objects as arguments. Espresso fires each action in sequence according to the given order and executes them in the main thread.

The ViewActions class provides a list of helper methods for specifying common actions. You can use these methods as convenient shortcuts instead of creating and configuring individual ViewAction objects. You can specify such actions as:

1. ViewActions.click(): Clicks on the view.
2. ViewActions.typeText(): Clicks on a view and enters a specified string.
3. ViewActions.scrollTo(): Scrolls to the view. The target view must be subclassed from ScrollView and the value of its android:visibility property must be VISIBLE. For views that extend AdapterView (for example, ListView), the onData() method takes care of scrolling for you.
4. ViewActions.pressKey(): Performs a key press using a specified keycode.
5. ViewActions.clearText(): Clears the text in the target view.

If the target view is inside a ScrollView, perform the ViewActions.scrollTo() action first to display the view in the screen before other proceeding with other actions. The ViewActions.scrollTo() action will have no effect if the view is already displayed.

1. Test your activities in isolation with Espresso Intents

Espresso Intents enables validation and stubbing of intents sent out by an app. With Espresso Intents, you can test an app, activity, or service in isolation by intercepting outgoing intents, stubbing the result, and sending it back to the component under test.

To begin testing with Espresso Intents, you need to add the following line to your app's build.gradle file:

dependencies {

androidTestImplementation('androidx.test.espresso:espresso-intents:3.1.0')

}

To use Espresso Intents, you first need to initialize it in the test setup via Intents.init()a> You may also want to use ActivityScenarioRule to launch the host activity before each test.

The test class shown in the following codes snippet provides a simple test for an explicit intent. It tests the activities and intents created in the Building Your First App tutorial.

private const val MESSAGE = "This is a test"

private const val PACKAGE\_NAME = "com.example.myfirstapp"

@RunWith(AndroidJUnit4::class)

class SimpleIntentTest {

/\* Instantiate an ActivityScenarioRule object. \*/

@get:Rule

var activityRule: ActivityScenarioRule<MainActivity> = ActivityScenarioRule(MainActivity::class.java)

@Before

fun setUp() {

Intents.init()

}

@After

fun tearDown() {

Intents.release()

}

@Test

fun verifyMessageSentToMessageActivity() {

// Types a message into a EditText element.

onView(withId(R.id.edit\_message))

.perform(typeText(MESSAGE), closeSoftKeyboard())

// Clicks a button to send the message to another

// activity through an explicit intent.

onView(withId(R.id.send\_message)).perform(click())

// Verifies that the DisplayMessageActivity received an intent

// with the correct package name and message.

intended(allOf(

hasComponent(hasShortClassName(".DisplayMessageActivity")),

toPackage(PACKAGE\_NAME),

hasExtra(MainActivity.EXTRA\_MESSAGE, MESSAGE)))

}

}

1. Test WebViews with Espresso Web

Espresso Web allows you to test WebView components contained within an activity. It uses the WebDriver API to inspect and control the behavior of a WebView.

To begin testing with Espresso Web, you need to add the following line to your app's build.gradle file:

dependencies {

androidTestImplementation('androidx.test.espresso:espresso-web:3.1.0')

}

When creating a test using Espresso Web, you need to enable JavaScript on the WebView when you instantiate the ActivityScenario object to test the activity. In the tests, you can select HTML elements displayed in the WebView and simulate user interactions, like entering text into a text box and then clicking a button. After the actions are completed, you can then verify that the results on the Web page match the results that you expect.

In the following code snippet, the class tests a WebView component with the id value 'webview' in the activity being tested. The typeTextInInput\_clickButton\_SubmitsForm() test selects an <input> element on the Web page, enters some text, and checks text that appears in another element.

private const val MACCHIATO = "Macchiato"

private const val DOPPIO = "Doppio"

@LargeTest

@RunWith(AndroidJUnit4::class)

class WebViewActivityTest {

@Test

fun typeTextInInput\_clickButton\_SubmitsForm() {

// Lazily launch the Activity with a custom start Intent per test

try (ActivityScenario.launch(withWebFormIntent())) {

onWebView().forceJavascriptEnabled()

// Selects the WebView in your layout.

// If you have multiple WebViews you can also use a

// matcher to select a given WebView, onWebView(withId(R.id.web\_view)).

onWebView()

// Find the input element by ID

.withElement(findElement(Locator.ID, "text\_input"))

// Clear previous input

.perform(clearElement())

// Enter text into the input element

.perform(DriverAtoms.webKeys(MACCHIATO))

// Find the submit button

.withElement(findElement(Locator.ID, "submitBtn"))

// Simulate a click via JavaScript

.perform(webClick())

// Find the response element by ID

.withElement(findElement(Locator.ID, "response"))

// Verify that the response page contains the entered text

.check(webMatches(getText(), containsString(MACCHIATO)))

}

}

}

1. Verify results

Call the ViewInteraction.check() or DataInteraction.check() method to assert that the view in the UI matches some expected state. You must pass in a ViewAssertion object as the argument. If the assertion fails, Espresso throws an AssertionFailedError.

The ViewAssertions class provides a list of helper methods for specifying common assertions. The assertions you can use include:

1. doesNotExist: Asserts that there is no view matching the specified criteria in the current view hierarchy.
2. matches: Asserts that the specified view exists in the current view hierarchy and its state matches some given Hamcrest matcher.
3. selectedDescendentsMatch: Asserts that the specified children view for a parent view exist, and their state matches some given Hamcrest matcher.

The following code snippet shows how you might check that the text displayed in the UI has the same value as the text previously entered in the EditText field.

fun testChangeText\_sameActivity() {

// Type text and then press the button.

...

// Check that the text was changed.

onView(withId(R.id.textToBeChanged))

.check(matches(withText(STRING\_TO\_BE\_TYPED)))

}

1. Run Espresso tests on a device or emulator

You can run Espresso tests from Android Studio or from the command-line. Make sure to specify AndroidJUnitRunner as the default instrumentation runner in your project.

## Testing UI for multiple apps

A user interface (UI) test that involves user interactions across multiple apps lets you verify that your app behaves correctly when the user flow crosses into other apps or into the system UI. An example of such a user flow is a messaging app that lets the user enter a text message, launches the Android contact picker so that the users can select recipients to send the message to, and then returns control to the original app for the user to submit the message.

This lesson covers how to write such UI tests using the UI Automator testing framework provided by AndroidX Test. The UI Automator APIs let you interact with visible elements on a device, regardless of which Activity is in focus. Your test can look up a UI component by using convenient descriptors such as the text displayed in that component or its content description. UI Automator tests can run on devices running Android 4.3 (API level 18) or higher.

The UI Automator testing framework is an instrumentation-based API and works with the AndroidJUnitRunner test runner.

1. Set up UI Automator

Before building your UI test with UI Automator, make sure to configure your test source code location and project dependencies, as described in Set up project for AndroidX Test.

In the build.gradle file of your Android app module, you must set a dependency reference to the UI Automator library:

dependencies {

...

androidTestImplementation('androidx.test.uiautomator:uiautomator:2.2.0')

}

To optimize your UI Automator testing, you should first inspect the target app’s UI components and ensure that they are accessible. These optimization tips are described in the next two sections.

1. Inspect the UI on a device

Before designing your test, inspect the UI components that are visible on the device. To ensure that your UI Automator tests can access these components, check that these components have visible text labels, android:contentDescription values, or both.

The uiautomatorviewer tool provides a convenient visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device. This information lets you create more fine-grained tests using UI Automator. For example, you can create a UI selector that matches a specific visible property.

1. To launch the uiautomatorviewer tool:
2. Launch the target app on a physical device.
3. Connect the device to your development machine.
4. Open a terminal window and navigate to the <android-sdk>/tools/ directory.
5. Run the tool with this command:

$ uiautomatorviewer

To view the UI properties for your application:

1. In the uiautomatorviewer interface, click the Device Screenshot button.
2. Hover over the snapshot in the left-hand panel to see the UI components identified by the uiautomatorviewer tool. The properties are listed in the lower right-hand panel and the layout hierarchy in the upper right-hand panel.
3. Optionally, click on the Toggle NAF Nodes button to see UI components that are non-accessible to UI Automator. Only limited information may be available for these components.
4. Ensure your activity is accessible

The UI Automator test framework performs better on apps that have implemented Android accessibility features. When you use UI elements of type View, or a subclass of View from the SDK, you don't need to implement accessibility support, as these classes have already done that for you.

Some apps, however, use custom UI elements to provide a richer user experience. Such elements won't provide automatic accessibility support. If your app contains instances of a subclass of View that isn't from the SDK, make sure that you add accessibility features to these elements by completing the following steps:

1. Create a concrete class that extends ExploreByTouchHelper.
2. Associate an instance of your new class with a specific custom UI element by calling setAccessibilityDelegate().

For additional guidance on adding accessibility features to custom view elements, see Building Accessible Custom Views. To learn more about general best practices for accessibility on Android, see Making Apps More Accessible.

1. Create a UI Automator test class

Your UI Automator test class should be written the same way as a JUnit 4 test class. To learn more about creating JUnit 4 test classes and using JUnit 4 assertions and annotations, see Create an Instrumented Unit Test Class.

Add the @RunWith(AndroidJUnit4.class) annotation at the beginning of your test class definition. You also need to specify the AndroidJUnitRunner class, provided in AndroidX Test, as your default test runner. This step is described in more detail in Run UI Automator tests on a device or emulator.

Implement the following programming model in your UI Automator test class:

1. Get a UiDevice object to access the device you want to test, by calling the getInstance() method and passing it an Instrumentation object as the argument.
2. Get a UiObject object to access a UI component that is displayed on the device (for example, the current view in the foreground), by calling the findObject() method.
3. Simulate a specific user interaction to perform on that UI component, by calling a UiObject method; for example, call performMultiPointerGesture() to simulate a multi-touch gesture, and setText() to edit a text field. You can call on the APIs in steps 2 and 3 repeatedly as necessary to test more complex user interactions that involve multiple UI components or sequences of user actions.
4. Check that the UI reflects the expected state or behavior, after these user interactions are performed.

These steps are covered in more detail in the sections below.

1. Access UI components

The UiDevice object is the primary way you access and manipulate the state of the 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 and Menu buttons.

It’s good practice to start your test from the Home screen of the device. From the Home screen (or some other starting location you’ve chosen in the device), 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 might get an instance of UiDevice and simulate a Home button press:

import org.junit.Before

import androidx.test.runner.AndroidJUnit4

import androidx.test.uiautomator.UiDevice

import androidx.test.uiautomator.By

import androidx.test.uiautomator.Until

...

private const val BASIC\_SAMPLE\_PACKAGE = "com.example.android.testing.uiautomator.BasicSample"

private const val LAUNCH\_TIMEOUT = 5000L

private const val STRING\_TO\_BE\_TYPED = "UiAutomator"

@RunWith(AndroidJUnit4::class)

@SdkSuppress(minSdkVersion = 18)

class ChangeTextBehaviorTest2 {

private lateinit var device: UiDevice

@Before

fun startMainActivityFromHomeScreen() {

// Initialize UiDevice instance

device = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation())

// Start from the home screen

device.pressHome()

// Wait for launcher

val launcherPackage: String = device.launcherPackageName

assertThat(launcherPackage, notNullValue())

device.wait(

Until.hasObject(By.pkg(launcherPackage).depth(0)),

LAUNCH\_TIMEOUT

)

// Launch the app

val context = ApplicationProvider.getApplicationContext<Context>()

val intent = context.packageManager.getLaunchIntentForPackage(

BASIC\_SAMPLE\_PACKAGE).apply {

// Clear out any previous instances

addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TASK)

}

context.startActivity(intent)

// Wait for the app to appear

device.wait(

Until.hasObject(By.pkg(BASIC\_SAMPLE\_PACKAGE).depth(0)),

LAUNCH\_TIMEOUT

)

}

}

In the example, the @SdkSuppress(minSdkVersion = 18) statement helps to ensure that tests will only run on devices with Android 4.3 (API level 18) or higher, as required by the UI Automator framework.

Use the findObject() method to retrieve a UiObject which represents a view that matches a given selector criteria. You can reuse the UiObject instances that you have created in other parts of your app testing, as needed. Note that 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 a property.

The following snippet shows how your test might construct UiObject instances that represent a Cancel button and a OK button in an app.

val cancelButton: UiObject = device.findObject(

UiSelector().text("Cancel").className("android.widget.Button")

)

val okButton: UiObject = device.findObject(

UiSelector().text("OK").className("android.widget.Button")

)

// Simulate a user-click on the OK button, if found.

if (okButton.exists() && okButton.isEnabled) {

okButton.click()

}

1. Specify a selector

If you want to access a specific UI component in an app, use the UiSelector class. This class represents a query for 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 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 text property Apps.

val appItem: UiObject = device.findObject(

UiSelector().className("android.widget.ListView")

.instance(0)

.childSelector(

UiSelector().text("Apps")

)

)

As a best practice, when specifying a selector, you should use a Resource ID (if one is assigned to a UI element) instead of a text element or content-descriptor. Not all elements have a text element (for example, icons in a toolbar). Text selectors are brittle and can lead to test failures if there are minor changes to the UI. They may also not scale across different languages; your text selectors may not match translated strings.

It can be useful to specify the object state in your selector criteria. For example, if you want to select a list of all checked elements so that you can uncheck them, call the checked() method with the argument set to true.

1. Perform actions

Once your test has obtained a UiObject object, you can call the methods in the UiObject class to perform user interactions on the UI component represented by that object. You can specify such actions as:

1. click() : Clicks the center of the visible bounds of the UI element.
2. dragTo() : Drags this object to arbitrary coordinates.
3. setText() : Sets the text in an editable field, after clearing the field's content. Conversely, the clearTextField() method clears the existing text in an editable field.
4. swipeUp() : Performs the swipe up action on the UiObject. Similarly, the swipeDown(), swipeLeft(), and swipeRight() methods perform corresponding actions.
5. The UI Automator testing framework allows you to send an Intent or launch an Activity without using shell commands, by getting a Context object through getContext().

The following snippet shows how your test can use an Intent to launch the app under test. This approach is useful when you are only interested in testing the calculator app, and don't care about the launcher.

fun setUp() {

...

// Launch a simple calculator app

val context = getInstrumentation().context

val intent=context.packageManager.getLaunchIntentForPackage(CALC\_PACKAGE).apply {

addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TASK)

}

// Clear out any previous instances

context.startActivity(intent)

device.wait(Until.hasObject(By.pkg(CALC\_PACKAGE).depth(0)), TIMEOUT)

}

1. Perform actions on collections

Use the UiCollection class if you want to simulate user interactions on a collection of items (for example, songs in a music album or a list of emails in an Inbox). 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 view that contains child UI elements.

The following code snippet shows how your test might construct a UiCollection to represent a video album that is displayed within a FrameLayout:

val videos=UiCollection(UiSelector().className("android.widget.FrameLayout"))

// Retrieve the number of videos in this collection:

val count = videos.getChildCount(

UiSelector().className("android.widget.LinearLayout")

)

// Find a specific video and simulate a user-click on it

val video: UiObject = videos.getChildByText(

UiSelector().className("android.widget.LinearLayout"),

"Cute Baby Laughing"

)

video.click()

// Simulate selecting a checkbox that is associated with the video

val checkBox: UiObject = video.getChild(

UiSelector().className("android.widget.Checkbox")

)

if (!checkBox.isSelected) checkBox.click()

1. Perform 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.

The following code snippet shows how to simulate scrolling down the Settings menu and clicking on an About tablet option:

val

settingsItem=UiScrollable(UiSelector().className("android.widget.ListView"))

val about: UiObject = settingsItem.getChildByText(

UiSelector().className("android.widget.LinearLayout"),

"About tablet"

)about.click()

1. Verify results

The InstrumentationTestCase extends TestCase, so you can use standard JUnit Assert methods to test that UI components in the app return the expected results.

The following snippet 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 const val CALC\_PACKAGE = "com.myexample.calc"

fun testTwoPlusThreeEqualsFive() {

// Enter an equation: 2 + 3 = ?

device.findObject(UiSelector().packageName(CALC\_PACKAGE).resourceId("two")).click()

device.findObject(UiSelector().packageName(CALC\_PACKAGE).resourceId("plus")).click()

device.findObject(UiSelector().packageName(CALC\_PACKAGE).resourceId("three")).click()

device.findObject(UiSelector().packageName(CALC\_PACKAGE).resourceId("equals")).click()

// Verify the result = 5

val result: UiObject2 = device.findObject(By.res(CALC\_PACKAGE, "result"))

assertEquals("5", result.text)

}

1. Run UI Automator tests on a device or emulator

You can run UI Automator tests from Android Studio or from the command-line. Make sure to specify AndroidJUnitRunner as the default instrumentation runner in your project.

**QUESTIONS:**

1. Explain the advantages of the computer system.

**Answer:**

* 1. **Computers can do the same task repetitively with the same accuracy.**
  2. **Computers do not get tired or bored.**
  3. **Computers can take up routine tasks while releasing human resources for more intelligent functions.**

**REFERENCE:**

1. https://developer.android.com/training/testing/fundamentals
2. <https://developer.android.com/training/testing/ui-testing>