LE/EECS 4443: Mobile User Interfaces (LAB)

Week 6: UI Testing (Espresso, UI Automator)

Shogo Toyonaga[1]

[1]Lassonde School of Engineering
York University

January 27, 2025

Table of Contents

Introduction

By the end of this tutorial, you will be able to...

1. Understand the theory (necessity) of software testing
2. Correctly apply software or usability testing
3. Implement UI Tests through Espresso or UI Automator
4. Generate synthetic workloads for your application

   Note: This will be very useful for (macro)benchmarking your application. We will discuss this in a future recitation.

Testing - Why?

- Software testing is a way to enforce **quality assurance**; it establishes a tangible agreement between a client & developer regarding the expected behaviour of the application.
- Often, Organizations will rely on international standards for software quality (e.g., **ISO/IEC 9126**) that affect factors such as requirements engineering, coding reviews, and integration guidelines.

Testing - Why?

- Above all else, the **goals** of software testing are to efficiently identify and patch as many errors as possible within budget & time constraints.
- **Note:** Software teams do their best to eliminate bugs and optimize for efficiency, however, it is unrealistic to expect every bug to be caught in one iteration.

Testing Strategies I

1. **Big Bang**: Applying testing using a post-hoc methodology. The system is only evaluated once all of its components have been implemented.
2. **Incremental**: Components of the system are procedurally tested using unit tests (e.g., JUnit). Following this, integration testing is performed to evaluate how groups of components interact with each other to identify problematic dependencies. Lastly, the entire system is evaluated.

   Note: Components of the system can be evaluated in varying orders with different tradeoffs: BFS, DFS, Bottom-Up, Top-Down

Testing Strategies II

Bottom-Up is easier to write/perform, however, the important results are slower to obtain.

Top-Down Important results can be obtained faster, however, stub development adds complexities into the working environment.

3. **Black/White-Box Testing**: Design of test-cases are contingent on the transparency of the system. White-box testing evaluates functional accuracy, maintainability, and reusability. Black-box testing evaluates documentation, availability, reliability, performance, and security.

Testing Strategies III

4. **Alpha/Beta Testing**: Alpha testing evaluates a prototype model of the software. The client is given the application to detect errors. Beta testing uses a more complete prototype that is given to multiple clients and users. Again, the client will detect the errors.

   Both strategies offer unsystematic testing that is difficult to reproduce. Extra care must be placed into ensuring the quality of reports from clients.

Testing Strategies: Which One to Use!?

- Identify your time & budget constraints!
  1. **Big Bang** has most likely been your go-to for coursework. It is okay for smaller projects with simple objectives, however, it makes debugging & refactoring far more difficult.
  2. **Incremental testing** is more preferred; continuous testing ensures that you can identify problems immediately and deal with them. It is however more time intensive.

Classification of Tests

- The three types of testing equivalence classes:
  1. **Happy:** A well-defined test case using known input that executes and produces an expected output. It does not guarantee handling of error conditions.
  2. **Boundary:** Synonymous with "edge" or "corner" cases. It tests the application under an extreme (min/max) operating parameter.
  3. **Exceptional:** Used to test the application under contract-breaking violation(s). We want to test the robustness of the applications error handling.
- A test-case can cover a wide range of happy and/or boundary classes. This means that you don't have to test every single combination of input value(s).
- At a minimum, you need one test case per invalid class.

Writing JUnit Tests in Android Studio

- Tools exist for the automatic execution of tests, preparation of registries, and reports. You should be familiar with JUnit (LE/EECS 1022, LE/EECS 2030)
- To generate a testing environment for a class (e.g., Model), simply highlight all of the code in the IDE and press CTRL + SHIFT + T.

Introduction

- We just discussed **software testing (LE/EECS 3311)**
- In LE/EECS 4443, we will add on **UI Testing**; it validates the state of the UI before and/or after interactions have taken place.

Introduction

The goals of UI Testing include:

1. Validating the **appearance / consistency** of UI elements & screens
2. Validating the **behaviour** of the application between the Model, View, and Controller. This includes **data & transition-based** changes that are onset by the user.

Types of UI Testing

1. Manual
   - Exploratory (Non-Exhaustive)
   - User Experience ($\approx \alpha/\beta$ Testing)
   - Scripted (Recall <u>OLD</u> LE/EECS 1022 Labs!)
2. Automated (Espresso, UI Automator)
3. Record & Replay (Espresso)

Jetpack Frameworks: An Easy Introduction

1. **Espresso**: "Espresso tests state expectations, interactions, and assertions clearly without the distraction of boilerplate content, custom infrastructure, or messy implementation details getting in the way."
2. **UI Automator**: "A UI testing framework suitable for cross-app functional UI testing across system and installed apps. The UI Automator APIs let you interact with visible elements on a device, regardless of which Activity is in focus, so it allows you to perform operations such as opening the Settings menu or the app launcher in a test device."

Espresso: Basics

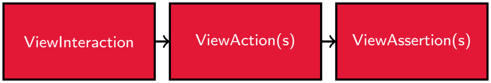
- Before we begin learning how to write & execute UI Tests, it is important to set up your testing environment correctly.
- On your device, under Settings > Developer Options, disable three animations known to cause failing tests:
  1. Window Animation Scale
  2. Transition Animation Scale
  3. Animator Duration Scale

Espresso: The Basics

- Assign ViewInteraction objects with onView(). These will be used to perform actions or validate state(s).
  1. Note: You can also use combination matchers.
  2. Note: You should use onData() instead of onView() for target views inside an AdapterView. onData() provides an entry point which is able to first load the adapter item in question, bringing it into focus prior to operating on it or any of its children.
- Call ViewActions such as perform(· · ·) to change the state(s) of the Activity or View(s).
- Call ViewAssertions such as check(· · ·) to check the state or values of Views / variables.

https://developer.android.com/training/testing/espresso/basics

Espresso: The Basics (Summary)



Case Study: Demo_Caesar (java > androidTest > MainActivityTest)

Writing an Espresso Test (Demo_Caesar) I

```
@LargeTest
@RunWith(AndroidJUnit4.class)
public class MainActivityTest {
    // Declare Views to Interact with
    private final ViewInteraction spinner = onView(withId(R.id.spinner_key));
    private final ViewInteraction user_input = onView(withId(R.id.user_input));
    private final ViewInteraction encrypt = onView(withId(R.id.encrypt));
    private final ViewInteraction decrypt = onView(withId(R.id.decrypt));
    private final ViewInteraction button = onView(withId(R.id.compute_button));
    private final ViewInteraction solution = onView(withId(R.id.computed_results
         ));
    // UiAutomator State
    UiDevice device = UiDevice.getInstance(InstrumentationRegistry.
         getInstrumentation());

    @Rule
    public ActivityScenarioRule<MainActivity> mActivityScenarioRule =
            new ActivityScenarioRule<>(MainActivity.class);
```

Writing an Espresso Test (Demo_Caesar) II

```
    /**
     * A simple test to demonstrate the forwards functionality of ROT-13
     */
    @Test
    public void rot13_simple_encrypt() {
        user_input.perform(typeText("Hello world!"));
        spinner.perform(click());
        onData(allOf(is(instanceOf(String.class)), is("13"))).perform(click());
        encrypt.perform(click());
        button.perform(click());
        solution.check(matches(withText("URYYB JBEYQ!")));
    }
```

Writing an Espresso Test (Demo_Caesar) III

```java
    /**
     * A test to demonstrate the effectiveness of onSaveInstanceState() and
         onRestoreInstanceState()
     *
     * @throws RemoteException
     */
    @Test
    public void switch_orientation_test() throws RemoteException {
        String quote = "Make peace with your mistakes and they'll turn to gold";
        user_input.perform(typeText(quote));
        encrypt.perform(click());
        button.perform(click());
        device.setOrientationLandscape();
        solution.check(matches(withText("PDNH SHDFH ZLWK BRXU PLVWDNHV DQG WKHB'
             OO WXUQ WR JROG")));
        device.setOrientationNatural();
        solution.check(matches(withText("PDNH SHDFH ZLWK BRXU PLVWDNHV DQG WKHB'
             OO WXUQ WR JROG")));
    }
}
```

Espresso: Closing Remarks

Some helpful resources to continue your learning include:

1. Espresso Basics
2. Espresso Cheat Sheet
3. Espresso Intents
4. Espresso Lists
5. Android Testing Playlist (Daniel Talks Code)

UI Automator: The Basics

- UI Automator and Espresso have feature overlap, however, Espresso has more native synchronization support. It is generally preferred to use Espresso over UI Automator for UI tests, however, we must use UI Automator for benchmarking purposes.
- UiDevice is used to access and perform operations on the device.
- UiObject2 is used to select a UI element for interaction.
- Assertions are used to evaluate the state of the application.

Writing UI Automator Code (Demo_Caesar) I

```
    /**
     * A benchmark to test ROT-13 on a short piece of plaintext
     */
    @Test
    public void ROT13_Encryption() {
        mBenchmarkRule.measureRepeated(
                "com.example.demo_caesar",
                // Put your list of metrics here....
                Arrays.asList(new StartupTimingMetric(), new FrameTimingMetric()
                        ),
                CompilationMode.DEFAULT,
                StartupMode.COLD,
                5,
                // Your benchmarking actions go here using UiAutomator to
                        control the device
                scope -> {
                    UiDevice device = scope.getDevice();
                    // Start the Activity and Wait
                    scope.startActivityAndWait();
                    // Attach Views to Variables (Controllers)
                    UiObject2 spinner, user_input, encrypt, decrypt, button,
                            solution;
                    spinner = device.findObject(By.res(scope.getPackageName(), "
                            spinner_key"));
```

Writing UI Automator Code (Demo_Caesar) II

```
                    user_input = device.findObject(By.res(scope.getPackageName()
                         , "user_input"));
                    encrypt = device.findObject(By.res(scope.getPackageName(), "
                         encrypt"));
                    decrypt = device.findObject(By.res(scope.getPackageName(), "
                         decrypt"));
                    button = device.findObject(By.res(scope.getPackageName(), "
                         compute_button"));
                    solution = device.findObject(By.res(scope.getPackageName(),
                         "computed_results"));
                    // Perform Actions
                    user_input.setText("Hello world!");
                    encrypt.click();
                    button.click();
                    return null;
                });
    }
```

UI Automator: Closing Remarks

Some helpful resources to continue your learning include:

1. Write automated tests with UI Automator

Conclusion

Remark

Thank you for your attention!
Questions?