Testing Your Android Applications

Alexander Nelson

November 17th, 2017

University of Arkansas - Department of Computer Science and Computer Engineering

The Problem

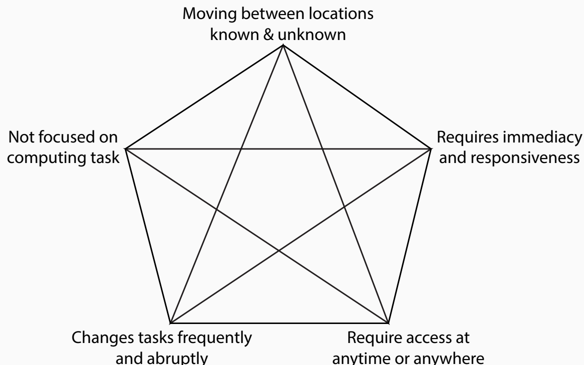
Testing Advantages

Testing your application provides the following advantages:

- Rapid feedback on failures
- Early failure detection in development cycle
- Safer code refactoring – optimize code without fear of regressing
- Stable development – minimize technical debt

Why is testing difficult?

Predicting user interaction patterns is difficult



Text from Reza B'Far - Mobile Computing Principles (2005)

Role of Workflows in App Development

Identify the major workflows within the application
Develop the application to direct users along those workflows

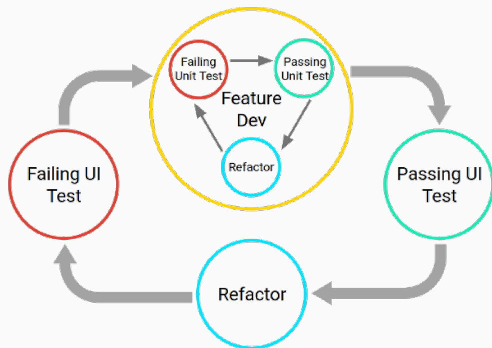
Advantages:

- Reduce number of cases your code must handle
- Reduce the search space for testing

Disadvantage:

- May limit the way a user can interact with the app
- Overcome this by listening to users/client and adding requested workflows

Iterative Development Workflow



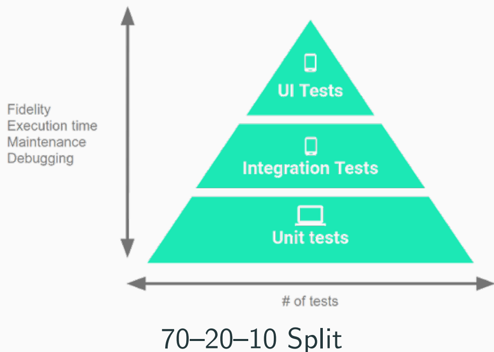
Two cycles – Unit tests for quicker development, UI tests for validation

Testing

Testing Pyramid

Your testing environment will likely include tests of different size

Should roughly approximate the testing pyramid



70–20–10 Split

Testing Pyramid

Small Tests – Fast and focused

- Address failures quickly
- Low fidelity, self-contained
- Passing all unit tests doesn't guarantee application will work

Medium Tests – Integration tests

- Integrate multiple components, and run on emulators or real devices
- Test how components work together to address intermediary issues

Large Tests – UI tests

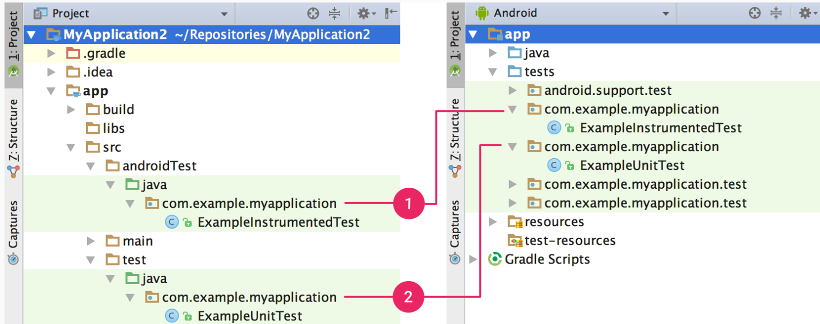
- Completes a UI workflow
- Ensure end-user tasks work as expected
- Slow and harder to diagnose issues

Types of Testing

There are three ways you can test the application:

1. Unit Tests – Inherit classes and check for correct output from given input
2. Instrumented Tests – Emulate user behavior with scripted input
3. Manually – Script user interactions and perform by hand

Testing Code in Android IDE



1) Instrumented Test

2) Unit Tests

Local Unit Tests

Local Unit Tests – Internal tests that are run inside the development JVM

Minimize execution time of tests that don't rely on Android Framework dependencies

Instrumented Tests

Instrumented Tests – Tests that run on hardware device or emulator

Have access to Instrumentation APIs

- Gives information about the app under test
- Allows control of application components within test code

Must have its own AndroidManifest file, but is auto generated on Gradle build

Creating test code

Easily create test code by following these steps:

1. Open java file with code to test
2. Click class to be tested and press "ctrl + shift + T"
3. A Create Test dialog appears, choose methods to generate and click "ok"
4. Choose the location for the test
    - androidTest for Instrumented Test
    - test for Local Unit Test

    Or create a Java file in the correct test source

Gradle Dependencies

Gradle Dependencies – Be sure that test library dependencies are included

- testCompile 'junit:junit:x.x' – JUnit 4 Framework
- androidTestCompile 'com.android.support:support-annotations:x.x.x'
- androidTestCompile 'com.android.support.test:runner:x.x'

Testing Tools

Android Testing Tools

Alternative to developing your own Unit and UI tests

Unit Tests – Roboelectric
http://robolectric.org/
Unit test framework that eases Android SDK plugin

UI Tests – Espresso
https://developer.android.com/training/testing/espresso/index.html
UI test framework

Roboelectric

Has Gradle and Maven plugins to build into your application

Allows creation of testcode with annotations

```
@RunWith(RobolectricTestRunner.class)
public class WelcomeActivityTest {

    @Test
    public void clickingLogin_shouldStartLoginActivity() {
        WelcomeActivity activity = Robolectric.setupActivity(WelcomeActivity.class);
        activity.findViewById(R.id.login).performClick();

        Intent expectedIntent = new Intent(activity, LoginActivity.class);
        Intent actual = ShadowApplication.getInstance().getNextStartedActivity();
        assertEquals(expectedIntent.getComponent(), actual.getComponent());
    }
}
```

Espresso

Espresso – UI Testing Framework

```
@Test
public void greeterSaysHello() {
    onView(withId(R.id.name_field)).perform(typeText("Steve"));
    onView(withId(R.id.greet_button)).perform(click());
    onView(withText("Hello Steve!")).check(matches(isDisplayed()));
}
```

Espresso Basics

Main API components include:

- Espresso – Entry point to interactions with Views
  - onView()
  - onData()
  - pressBack()
- ViewMatchers – Collection of objects that implement Matcher interface
  - Used to find a view within the current view hierarchy
- ViewActions – Objects that can be passed to the perform method (e.g. Click)
- ViewAssertions – Objects that can be passed to the check() method
  - Similar to Assert methods in other tests – Fail if value is unexpected

Finding a View

For most views, you can use the R.id of the view to match
onView(withId(R.id.view_to_find))

However, not all views will have a unique R.id
(e.g. Inflated views for a listview)
You can look for other unique properties of the views
onView(allOf(withId(R.id.view_to_find), withText("Hello!")))
or
onView(allOf(withId(R.id.view_to_find), not(withText("Hello!"))))

Finding a View – Considerations

When finding a view:

- All views that a user can interact with should contain descriptive text or a content description
  - If not unique, it will be an accessibility bug, as well as being an issue with your testing
- Use the least descriptive matcher that is unique so that the test framework doesn't do more work than necessary
- If the target view is inside an AdapterView (e.g. ListView) onView() may not work
  - onData() method should be used instead

Performing an Action on a View

Once you have found the view, you will want to have the framework perform an action

You can perform instances of the ViewAction class using the perform() method

Examples:

- onView(...).perform(click());
- onView(...).perform(typeText("Hello"));
- onView(...).perform(scrollTo()); – For getting to views inside a scroll view

More than one action can be performed in a call

onView(...).perform(scrollTo(),click());

Checking View Assertions

After interaciton, check that proper transformations have occurred

check() method – Used to check assertions on selected view

Example:
onView(...).check(matches(withText("Hello")));

Creating your first test

Imagine a simple activity with a Button (btn) and a TextView (tv)
When button is clicked the content of the TextView changes to "Hello"

Test this by:

1. onView(withId(R.id.btn)).perform(click());
2. onView(withId(R.id.tv)).check(matches(withText("Hello")));

Adapter Views

Adapter Views may not have unique R.ids for each view
Use onData() instead of onView() for matching

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

Checks through data in current activity to find instances of a certain class (String)
Looks for a view with a specific value("Hello")
Performs click() on that view

Using the Espresso Recorder

The espresso recorder is built into Android Studio
Allows developer to create tests by performing actions

Caveats:

- Does not always create tests that will pass
- Does not always create tests in optimal ways