ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 18 — February 12, 2013

Patrick Lam *version 1*

Testing for Android

We'll also start talking about testing for Android. It's easy if you test classes that don't use Android. This is good program design anyway: split the Android-dependent parts from the general parts. Do that, when possible.

For the purpose of this course, you could put the step recognition code in a separate class and call it from the Activity. That would be a first step towards a Model-View-Controller design. (For more on that, and other designs, take ECE452).

However, if you want to test the Android-specific parts of your app, you'll need something to call those parts, because Android apps are event-driven. We'll talk about what Android provides for test automation. Note that this is not really a unit test. Despite its name, JUnit isn't just for writing unit tests. You can also use it for integration tests.

What To Test. You can test these aspects of your app manually; however, it's better to have a test script for them[1].

Orientation change:

- is screen redrawn correctly?
- did you lose any state?

Configuration change (e.g. adding a keyboard):

- again as for orientation change;
- do you handle the new device properly?

Battery life:

- don't hog the battery (out of scope for us).

External resources:

- how does your app behave when it doesn't have necessary resources, e.g. GPS?

We'll continue by talking about how to test Android apps.

[1]`http://developer.android.com/tools/testing/what_to_test.html`, accessed 7Feb13.

Android Activity Unit Tests. It is allegedly possible to test an `Activity` in isolation using the `ActivityUnitTestCase`. That would be more in the spirit of unit tests. However, I couldn't actually find useful tutorials about how to do that, so I'll just say that it's beyond the scope of this course.

Android Integration Test Cases.

The basic idea here is that a test case is a set of program inputs and expected program outputs. While it is possible to do this manually, that would restrict our ability to inspect program state, unless the program had debug statements (which consume resources and are tedious to manually verify).

We'd instead like to automate Android testing. Use `ActivityInstrumentationTestCase2`[2].

Creating an Android integration test case takes quite a few steps. Basically, you create a new project and make it call the app that you're trying to test. The test case and the app are closely tied together.

A general reference to Android testing is here:

`http://developer.android.com/tools/testing/activity_testing.html`

It is too generic, though. A more useful (but overly-long) tutorial is here:

`http://developer.android.com/tools/testing/activity_test.html`

However, that tutorial is too long. I'll give you the highlights.

Creating a Test Project. After you have your main app project, you need a test project as well. Create a new project with File > New > Other and choose "New Android Test Project". If I called my original package `ca.patricklam.ece155.A4`, I should call the test package `ca.patricklam.ece155.A4.text`.

You should now have both the Android activity and the test project. The test project should currently be empty.

Creating a Test Project Class. Next, you need to actually create a test class and populate it with test methods. This is a new class that you create in the test project, e.g. `DatePickerActivityTest`. Add it with File > New > Class.

- The superclass needs to be `ActivityInstrumentationTestCase2<DatePickerActivity>`.

In that class, you'll need a constructor, a setup method, and test methods.

[2]This is terrible naming!

Constructor. For technical reasons, your test class needs a constructor with the package name and class of the activity under test:

```
public DatePickerActivityTest() {
  super("ca.patricklam.ece155.A4", DatePickerActivity.class);
}
```

Setup method. Unlike the constructor, the setup method may run many times. It needs to set up the infrastructure for the test cases. Here's an example:

```
// fields for objects under test
private DatePickerActivity activityUnderTest;
private TextView monthViewUnderTest;
private Button okButton;
private Instrumentation instrumentation;

// name must be setUp() for JUnit 3, as used by Android.
@Override
protected void setUp() throws Exception {
  super.setUp();

  // if this is missing, any key events you send will be ignored
  setActivityInitialTouchMode(false);
  activityUnderTest = getActivity();
  monthViewUnderTest = (TextView) activityUnderTest.findViewById
                                   (ca.patricklam.ece155.A4.R.id.month);
  okButton = (Button) activityUnderTest.findViewById
                                   (ca.patricklam.ece155.A4.R.id.button);

  // lets you control the test object
  instrumentation = getInstrumentation();
}
```

Test Cases. Now you can use JUnit infrastructure to write test cases. This is a bit tricky due to the UI thread issue we encountered last time. Anything you do to the activity UI has to be in a `runOnUiThread()` block, just like with the `Timer`. You can also send keys and wait for the activity to finish chugging along. Finally, you'll want assertions.

```
public void testBadMonth() {
  // clear state of month textview
  activityUnderTest.runOnUiThread(
    new Runnable() {
      public void run() {
        monthViewUnderTest.setText("");
        monthViewUnderTest.requestFocus();
      }
    });

  // try to insert "13" in the month
```

```
    this.sendKeys(KeyEvent.KEYCODE_1);
    this.sendKeys(KeyEvent.KEYCODE_3);
    this.sendKeys(KeyEvent.KEYCODE_ENTER);

    // let it process the key events
    instrumentation.waitForIdleSync();

    // payload: check to see that the field rejects the entry
    assertFalse("text field not 13",
                monthViewUnderTest.getText().toString().equals("13"));
  }
```

At this point, you can test your test and see if it works. If you're using Test-Driven Development, you'd build the test first, see that it fails, and then fix the code. To run it, just select the test project and run that.

Other Things to Test. The tutorial also suggests verifying the state of the Activity before you do anything. That's a good idea. Also, for 1 point on the assignment, I'm asking you to write a test case that causes a phone rotation and checks that the right thing happens. You can consult the rest of the tutorial for information on how to do that. You'll want to use the instrumentation to send broadcast events to the phone.

Software Testing (more general view)

We offer a complete course on software testing, ECE453. Here are some highlights.

Testing attempts to verify the functionality of your software. A key notion in testing is that of *coverage*, which makes sure that you're not missing any corner cases. When you're testing, you'll run *test suites* consisting of *test cases.*

A *test case* contains:

- what you feed to software; and
- what the software should output in response.

You can organize the collection of test cases you need to write according to a *test plan.*

Testing versus correctness. Common wisdom used to be that you could only find defects using testing, and not prove correctness, which would require exhaustive testing. However, there have been recent research advances which make it possible to prove correctness using testing.

Levels of Testing

Testing is often performed at several different levels:

- *Unit tests* are low-level tests which verify the functionality of a single class (or unit) at a time.

- *Integration tests* combine more than one unit and verify the functionality of the interfaces between components.
- *System tests* run on the entire system, after it has been integrated, and verify that everything works right.

Unit Testing

We will discuss principles behind unit testing and how and when to best deploy unit testing. We'll provide definitions of what people usually mean when they say unit tests.

1. Test small parts—units—of a software system (method, class, set of classes) independently.
2. Specify the desired behaviour of the unit using tests.

The following concepts are related to unit testing: test-driven development[3] (TDD), which is currently the most-advocated way to write unit tests; mock objects; and automation. Some basic properties of unit tests:

- focus on the unit being tested; for instance, don't depend on a database, network, or other external resources;
- easy to run by anyone (e.g. by setting them up as JUnit tests): they must incur no setup costs to run nor require user input; they can therefore serve as regression tests;
- easy to write (a few minutes per test) and focus on one aspect of behaviour; they should tell you something about the unit.

Unit tests come with a lot of baggage and people telling you how to do things, e.g.:

- specify and document the requirements of the unit;
- test behaviour, not state, and use mock objects to verify behaviour;
- some say that unit tests ought to be created during development (TDD) and written first, even before the code to be tested exists.

Regression Testing

Regression testing refers to any software testing that uncovers errors by retesting the modified program (Wikipedia). Often refers to large suites of test cases which detect regressions:

- of a bug fix that a developer has proposed.
- of related and unrelated other features that have been added.

Attributes of Regression Tests. Regression tests usually have the following attributes:

- **Automated**: no real reason to have manual regression tests.

[3] `http://radio-weblogs.com/0100190/stories/2002/07/25/sixRulesOfUnitTesting.html`

- **Appropriately Sized**: suites that are too-small will miss bugs; suites that are too-large take too long to run. Optimally, we want to run tests continuously.
- **Up-to-date**: ensure that tests are valid for the version of program being tested.

Industry Processes

Good software companies have certain testing features that help them deliver good quality software. The following practices are used in industry:

- **Unit Tests:** Each class has an associated unit test. If you change a class, you must also modify the unit test.
- **Code Reviews:** Each branch in the central version control system has owners. To commit code to that branch, you must have your code reviewed and approved by one of the owners. This ensures code quality.
- **Continuous Builds:** There is often a machine that continuously checks out and tests the latest code. All unit and regression tests are run and the status is made public to the team. The status contains information about the whether the code was built successfully, whether all unit and regression tests passed, and a list of the last few commits that were made to the branch. Social pressure: if you break something, everyone knows about it :).
- **QA Team:** If all tests have passed, a QA team will look for additional bugs.
- **Release:** Once QA has approved a build, it is released for use.

Bug Tracking Systems

We talked about these systems in the context of debugging. If you don't write it down, you'll forget it. Defect tracking systems keep lists of defects in a database (these days, often web-accessible). Tracking systems keep track of reported defects and their confirmations; who is assigned to fix the defect; and defect status. Close a defect by changing its status to "resolved".

Bugzilla[4] is one popular web-based defect tracking system.

Case Study: QA for Android Games

Let's make it real by throwing in a case study of Android game QA[5]

Test On Real Devices. Emulators are good, but they're not always perfect. If you really care about quality, or if you want to handle sensor input, you will need to test on real devices. You need to figure out how many devices to test on; many developers test on a small number (e.g. 5) of devices, which will cover most markets. Apparently the Asian market is very fragmented, though, and companies maintain matrices of devices (tablet vs phone, high-res vs low-res, which GPU).

[4] `http://www.bugzilla.org`

[5] `http://techcrunch.com/2012/06/02/android-qa-testing-quality-assurance`, accessed 12Feb13.

Test Continuously. Pocket Gems makes Android games. They create and perform tests, and then offshore the rest of the testing. The offshore testing team files bugs in the bug tracker, which go back to the US.

Testing is tightly integrated with development. You can't test at the end and hope to succeed.

They also run a final test phase before shipping code. That phase is expensive and thorough, checking memory, performance, and device compatibility.