Engineering Design w/Embedded Systems

Lecture 18—Android JUnit Tests; General View of Testing

Patrick Lam
University of Waterloo

February 11, 2013

Part I

Testing for Android

Avoiding Android-Specific Testing

It's easy if you test classes that don't use Android.

Do that, when possible.

Example:

- put step recognition code in separate class; and
- call that code from your main Activity.

We'd be progressing towards a Model-View-Controller (ECE452) design.

Why Android Testing is Hard

Recall that Android apps are **event-driven**.

Need something to produce events for you.

This is not really unit testing, more like integration tests.

What To Test[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?

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

Android Activity Unit Tests

This is allegedly possible using `ActivityUnitTestCase`.

More like unit tests than what we'll see next.

There is no useful documentation on the Internet, apart from Javadoc.

Beyond the scope of this course.

About Test Cases

Basic idea: a test case is:

- set of program inputs; and
- expected outputs.

We'd like to automate this (why?)

Android Integration Test Cases

We will use a `ActivityInstrumentationTestCase2`.

Outline:

- Create a new app which integrates with the app under test.
- Test app contains test cases.

In class, I'll demo the steps of creating a test app. The instructions are in the PDF notes.

Part II

General Info On Software Testing

Software Testing: Highlights

We offer a complete course on software testing, ECE453.

- *Testing*: verify the functionality of your software.
- Key notion: *coverage*, ensure you're not missing any corner cases.
- You run *test suites* consisting of *test cases*.

Test Cases

A *test case* contains:

- input; and
- associated expected output.

Organize test cases according to a *test plan*.

Testing vs correctness

Historically: only find defects using testing, and not prove correctness, which would require exhaustive testing.

However, recently in research: prove correctness using testing.

Levels of Testing

- *Unit tests*: low-level tests; verify the functionality of a single class at a time.
- *Integration tests* combine more than one unit; verify functionality of interfaces between components.
- *System tests*: run on entire system, after integration; verify that everything works right.

Unit Testing: Related concepts

- test-driven development[2] (TDD);
- mock objects; and
- automation.

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

Unit Testing: Basic Properties

- focus on the unit being tested (no external dependencies)
- easy to run by anyone (e.g. by setting them up as JUnit tests);
- easy to write (a few minutes per test);
- focus on one aspect of behaviour;
- they should tell you something about the unit.

Unit Testing: Baggage

Unit tests should:

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

Regression Testing

"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.

Regression Testing: Attributes

- **Automated**: no reason to have manual regression tests.
- **Appropriately Sized**: suites that are too-small will miss bugs; suites that are too-large take too long to run.
- **Up-to-date**: ensure that tests are valid for the version of program being tested.

Tests in Practice

Practices 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 an owner. Ensures code quality.
- **Continuous Builds:** A machine continuously checks out and tests the latest code.
  All unit and regression tests are run; status made public.
  Social pressure: if you break something, everyone knows!
- **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

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".
Popular web-based example: Bugzilla[3].

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