Lecture 16 − Testing for Android

Jeff Zarnett & Patrick Lam
`jzarnett@uwaterloo.ca` & `p.lam@ece.uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

December 29, 2014

Part I

Code Coverage

How well are our test cases are testing the program?

Key concept: Code Coverage

What lines of the program are executed during our tests?

Difficult to determine when there is conditional logic.

Conditional: a statement evaluated to true or false.

The evaluated value is used to decide what the next step is.

Example: `if (x == 0) { foo(); } else { bar(); }` .

There are some different definitions for coverage:

- **Statement Coverage**
- **Branch Coverage**
- **Multiple Condition Coverage**

A line is "covered" if that line is executed during the course of the test cases.

A branch is covered if that branch is executed during the course of the test cases.

In multiple condition coverage, a class or module is covered if all combinations of the conditionals within it are tested.

We will focus on line coverage, mostly.

Getting 100% statement coverage is generally unrealistic.

Even if we covered all lines, we can still have bugs.

Dividing two numbers may work in the test cases.
result in a divide by zero error if the user puts in 0.

We say code is "dead" if it can never be run.

Sometimes the compiler identifies it for you.

Other times, it's only revealed by execution.

Dead code does no harm, but effort might be wasted on maintaining it.

Remove it altogether.

As test cases are added we get diminishing returns.

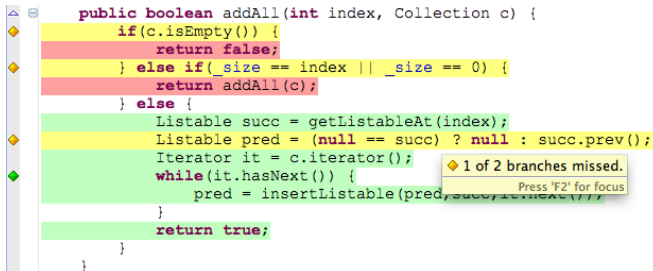The first test is all "new" lines.

Additional cases might have some overlap.

We will examine a plug-in for Eclipse called EclEmma.

When launched, EclEmma runs the selected unit tests and then analyzes the results.

EclEmma will colour-code each line in either:

- **Red** – the line was not executed;
- **Yellow** – a conditional where only one branch was taken (e.g., an if was always evaluated as false); or
- **Green** – the line was executed, or in the case of a conditional, both branches were taken.

```java
    public boolean addAll(int index, Collection c) {
        if(c.isEmpty()) {
            return false;
        } else if(_size == index || _size == 0) {
            return addAll(c);
        } else {
            Listable succ = getListableAt(index);
            Listable pred = (null == succ) ? null : succ.prev();
            Iterator it = c.iterator();
            while(it.hasNext()) {
                pred = insertListable(pred,succ,it.next());
            }
            return true;
        }
    }
```

◆ 1 of 2 branches missed.
Press 'F2' for focus

Using the coverage data, you can decide if your tests are effective in testing the logic of the code.

Too many red lines in the project means more tests (or better tests) are needed.

Each line of a method containing logic or a statement is colour-coded:

- Red: if the line was not executed;
- Yellow: if the line is a conditional where only one branch was taken; or
- Green: if the line was executed, or in the case of a conditional, both branches were taken.

```
public double computeValue(int input1, int input2) {
  if (input1 < 0) {
    throw new IllegalArgumentException(``Input 1 may not be negative'');
  }

  if (input2 == -1) {
    return 0;
  } else if (input2 <= 1) {
    return 1;
  }

  int result = (input1 * input2);
  result += (input1 / input2);

  return result;

}
```

Test case 1: computeValue(7, -1)
Test case 2: computeValue(0, 1000)

Part II

Testing for Android

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.

Recall that Android apps are event-driven.

Need something to produce events for you.

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

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.

This is allegedly possible using
`ActivityUnitTestCase`.

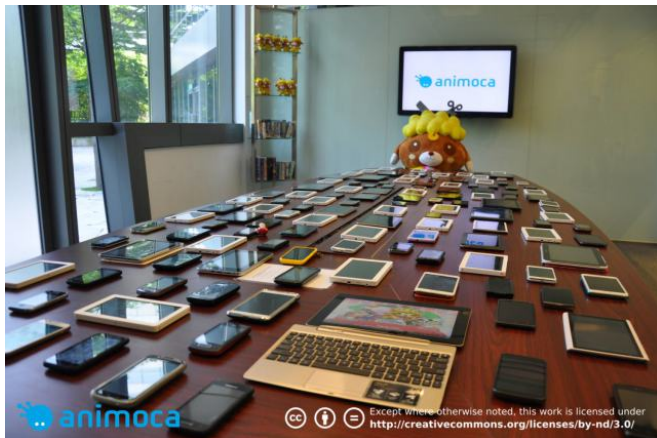More like unit tests than what we'll see next.

There is no useful documentation on the Internet, apart from Javadoc.
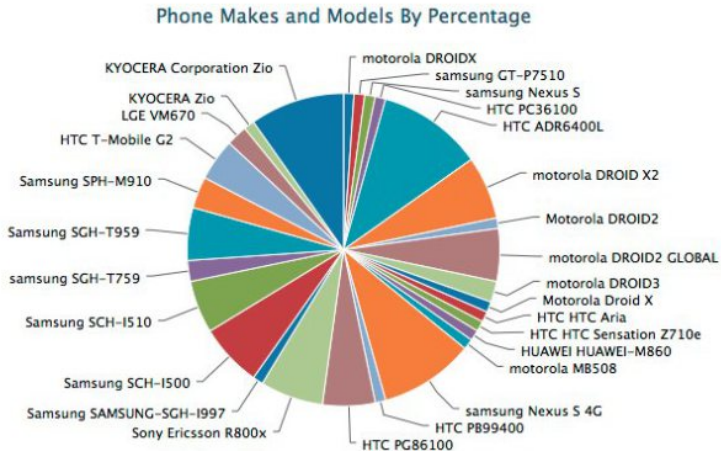
Beyond the scope of this course.

Part III

Case Study: QA for Android Games

Now we will examine a real situation: Android Game Testing

The biggest challenge is: fragmentation.

Phone Makes and Models By Percentage

Tip 1: **Test on Real Devices**.

Emulators are good, but not perfect.

Sensors and touch interface hard to emulate.

Tip 2: **Choose your Devices**.

How many is enough?
Some use a small number (5-12); others > 40.

Criteria: phone vs tablet, highres vs lowres screen, GPU.

Individual devices (e.g., Galaxy S3).

Tip 3: **Know what NOT to test**.

Decide what devices you will not support.

Choose to say no to small, slow, or outdated devices.

Tip 4: **Test Continuously**.

Pocket Gems creates test, but offshore the rest of the testing.

The offshore team files bugs.

Tests created concurrently with development.

It's not enough to just check the software.
Battery life, memory use, and performance must be tested.

Testing is tightly integrated - no "big bang" testing.

Run a final, extra thorough test phase before release.