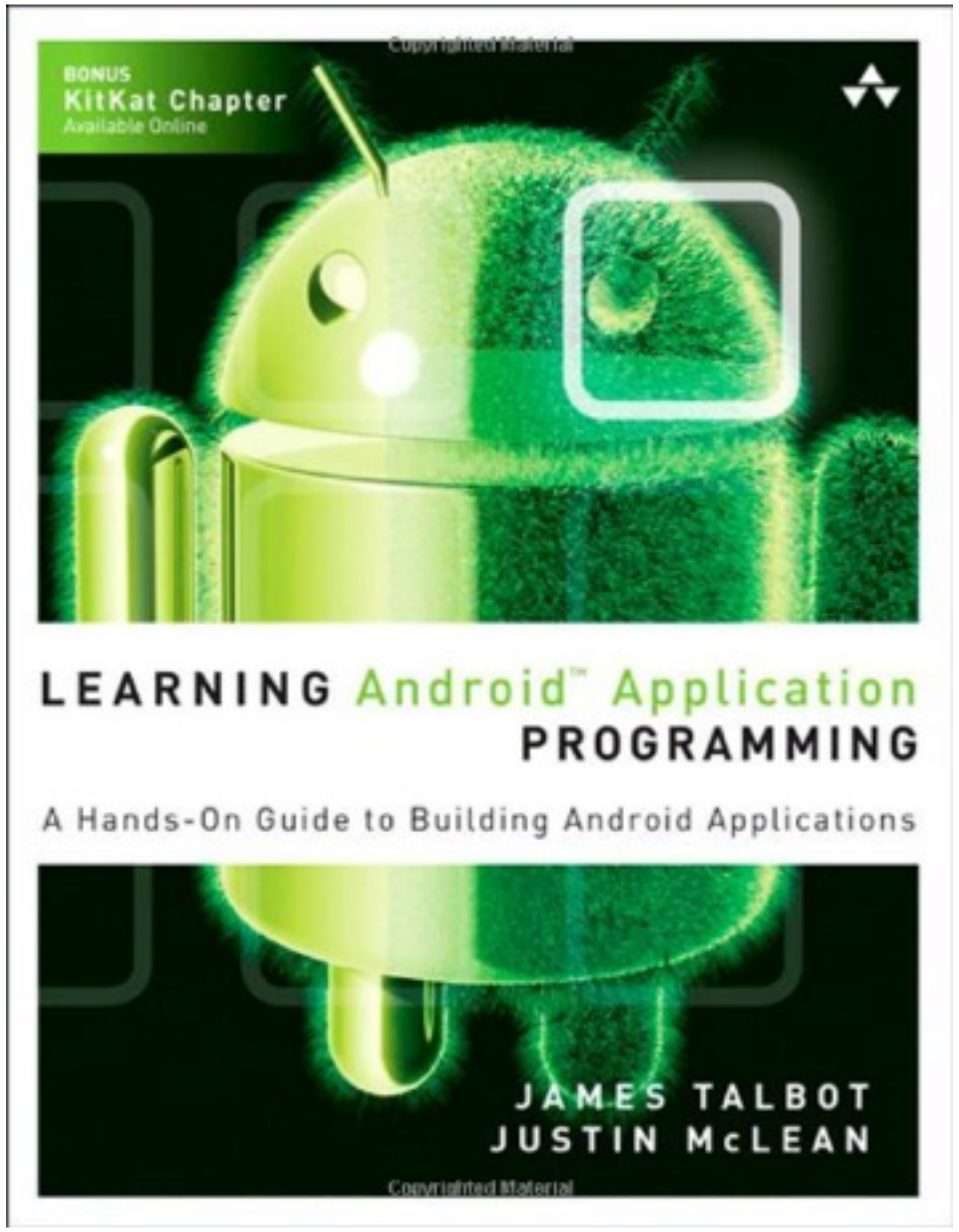
Testing Android Applications the Right Way - Part I

Justin Mclean
Email: justin@classsoftware.com
Twitter: @justinmclean

AnDevCon
The Android Developer Conference

Who am I?

- Freelance developer for 20 years
- Co-author of a book on Android Development
- Involved in the Apache Flex project
- Release manager for Apache Flex and Flex Unit
- Run the IoT Sydney meetup user group


BONUS
KitKat Chapter
Available Online
LEARNING Android™ Application PROGRAMMING
A Hands-On Guide to Building Android Applications
JAMES TALBOT
JUSTIN McLEAN

Learning Android Application Programming

James Talbot and Justin Mclean

Book Discount

- 35% off from InformIT.com
- Use discount code MCLEAN2931
- Also available form Amazon and O’Reilly Safari Books online

Testing the Right Way Part II

- Right after the coffee break 10am - 11:15 am
- Cover Android JUnit extensions, activity lifecycle testing and continuous integration
- Not talking about testing frameworks
- Slides for Part I and Part II are available
- https://github.com/justinmclean/AnDevCon2014/

Testing Methods

- Lots different things you need to test
- Many ways, tools and methods to test an application
- Different methods have different advantages and disadvantages
- Limited time and resources
- How do you effectively test your application?

Why People Don’t Test

- Time / budget pressures
- Frameworks are complex to set up and use
- Technical debt in existing code
- Lack of experience or training

Benefits of Testing

- You know your application works and is reasonably bug free
- Can make large changes and know you’ve not broken anything
- Know when your code is right quality
- Done right it adds little or no time overhead to a project and saves time in the future

The Emulator

- Can be slow especially startup
- Use snapshots and intel haxm to speed up
- Swap flavours of Android regularly
- Can be awkward to test sensors / phone calls / SMS
- Slow to reproduce results as it involves manual labour
- Consider using genymotion

Devices

- Fast to test
- Tests not reproducible
- Only need a few number of devices to get good coverage
    - low res/low power/small screen
    - modern phone (nexus good choice)
    - common phone (S3/S4 good choice)
    - 7” or 10” Tablet

The Monkey

- Fast and easy to use
- Acts like a hyperactive 4 year old randomly pushing buttons, changing volume, switching apps, rotating screen etc etc
- It will find bugs!
- Tests are reproducible via seed number

Command Line

- Run via command line:
  adb shell monkey -p <package> -v <no events>
- Change share % of the each type of event
- -s <seed no> will run the same sequence again

Locale Testing

- Best to do so early on as it’s more expensive to make changes down the track
- Use google translate toolkit and pick a few languages
- Japanese or Chinese, Hebrew or Arabic, Icelandic or German or Welsh are good choices for particular issues
- Shows were your UI needs tweaking or isn’t flexible

Lost in Translation

```
<string name=“app_name">
            あなたのバイクに</string>
<string name=“title_activity_timer">
            タイマー</string>
<string name=“title_activity_settings">
            設定</string>
<string name=“hello_world">
          こんにちは、世界h</string>
<string name="menu_settings">設定</string>
```

JUnit 4

- Run tests quickly again and again
- You know when you’ve broken something
- Saves you time and money
- Can take time to write and get right
- Can be hard to retrofit

IDE Support

- Can create JUnit projects
- Have tests in separate project means it not accidentally shipped as part of application
- Run and see test pass or fail in the IDE

JUnit 4

- Use annotations @Before @After and @Test
- @Before method run before each test
- @After run after each test
- @Test is a method called by the test runner
- Assert static methods

@Before and @After

```
@Before
public void setUp() throws Exception {
    timer = new TimerState();
}

@After
public void tearDown() throws Exception {
    timer = null;
}
```

@Test

```
@Test
public void initialState() {
    assertTrue("ElapsedTime is 0",
            timer.elapsedTime() == 0);
    assertTrue("Seconds is 0",
            timer.seconds() == 0);
    assertTrue("Minutes is 0",
            timer.minutes() == 0);
    assertTrue("Hours is 0”,
            timer.hours() == 0);
    assertTrue("IsRunning is false”,
            timer.isRunning() == false);
}
```

Real World Testing

- Real classes may be state or even time dependant. How do you easily test those?
- May need to refract code to make it easier to test
- Easier to test small class and methods

Real World Testing

```
@Test
 public void timerStarted() {
     timer.start();
     assertTrue("IsRunning is true”,
         timer.isRunning() == true);
     assertTrue("ElapsedTime > 0",
         timer.elapsedTime() > 0);
 }
```

Testing is Fast!

- Previous test fails as not enough time has passed between the two calls
- One workable solution would be to add a small delay either a tight loop or Thread.sleep
- But still can’t easily test the display method
- A better solution would be to extend the class you’re testing and return a fake value

Real World Testing

```
@Test
public void timerStarted() {
    timer.start();
    assertTrue("IsRunning is true”,
         timer.isRunning() == true);
    timer.setElapsedTime(1000);
    assertTrue("ElapsedTime > 0”,
         timer.elapsedTime() > 0);
}
```

Extend and Fake

```
private long fakeTime = -1;

public void setElapsedTime(long time) {
    fakeTime = time;
}

@Override
public long elapsedTime() {
    if (fakeTime == -1) {
        return super.elapsedTime();
    } else {
        return fakeTime;
    }
}
```

Even Faster Tests

```
@Test
public void oneMinute() {
    timer.setElapsedTime(1000 * 60);
    timer.stop();
    assertTrue("One minute has passed”,
     timer.display().equals("0:01:00"));
}
```

Testing Classes

- How do you test internals?
- Can rewrite code or make it more testable OR write test class to peek inside
- Test code can be different quality than production code

Test Coverage

- Want to try and cover as much code as possible
- Want to try an cover all branches in code
- May not always be possible
- Always a trade off vs time and effort
- Having a single test is better than have none and have 50% coverage is far better than 10%

Refactor / Simplify

- Remove as much Android / system calls as you can
- Simplify - each classes should do one thing
- Often has unexpected benefits
- Writing testable code makes testing easier

Removing System Calls

```
public class Time {
    public long now() {
        return System.currentTimeMillis();
    }
}
```

Stubbing / Mocking

- Can speed up tests
- Can exactly specify state of application
- Can easily test errors
- Increased cost to maintain - double the amount of code changes

Simple Mocking

```
public class SettableTime extends Time {

    public long time;

    @Override
    public long now() {
        return time;
    }
}
```

Mocking via Interfaces

- Refactor class to generate interface
- Extend class and implement interface
- Make each method behave how you want
- Good for slow calls (web service) or testing faults / error conditions

Testing isn't perfect

- Unit testing will not find all bugs even if you think you have 100% coverage
- Users will use your application in way you didn't expect
- Tests can give you a false sense of security
- Tests sometime give false positives

Good Testing

- Keep test simple and easy to understand
- Make tests fast
- Test code doesn’t have to be the same style or quality as application code
- Only test what you can in your time and budget
- Use the most effective testing method

Testing Frameworks?

- Can force you to use a single way of testing
- But can make testing easier
- Can be a good way to start if you not familiar with Unit testing and how to write testable code
- Take a look Robotium or Roboelectic or any of the others out there

Questions?

- Email me justin@classsoftware.com
- Twitter @justinmclean

Book Discount

- 35% off from InformIT.com
- Use discount code MCLEAN2931
- Also available form Amazon and O’Reilly Safari Books online