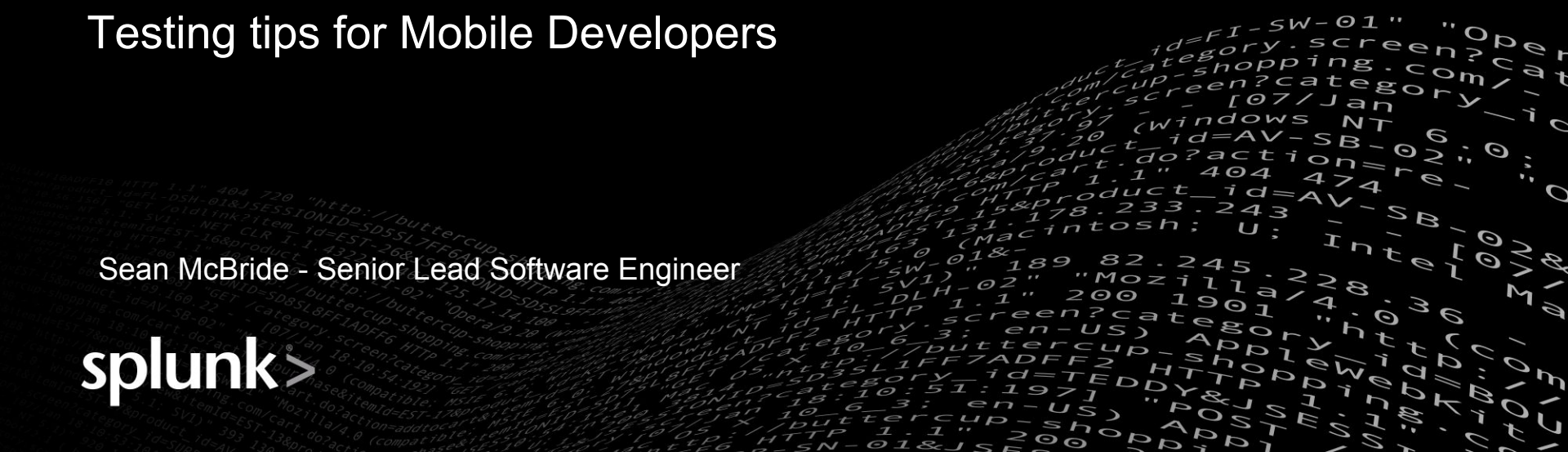
Mobile Dev Testing

Testing tips for Mobile Developers

Sean McBride - Senior Lead Software Engineer

splunk>



Who am I?

Sean McBride

- I got my start working on C/C++ client side code & speech recognition
- I work at Splunk in Boulder On the mobile team & lead a team of 5 developers working on the VictorOps Mobile applications (Android & iOS)
- Many of you might know taken a class with Greg Greenstreet. He's my boss' boss' boss
- Been doing this for 25+ years, leading teams for over half of that

Why the heck do we write tests?

1 - Writing tests helps you write better code

- I ALWAYS find some issue when writing tests
  - (could be I am just really bad at this whole dev thing or maybe...)
- Want to be sure you got it right?
  - Write a test
- Thinking about how to test something helps you understand it
  - Especially unit tests

2 - Code is a living thing

- Pull up some code you wrote a couple weeks ago and change it
  - Chances are you have to really think hard about it
- You aren't the only one with your hands in the code
  - Someone else may need to change something

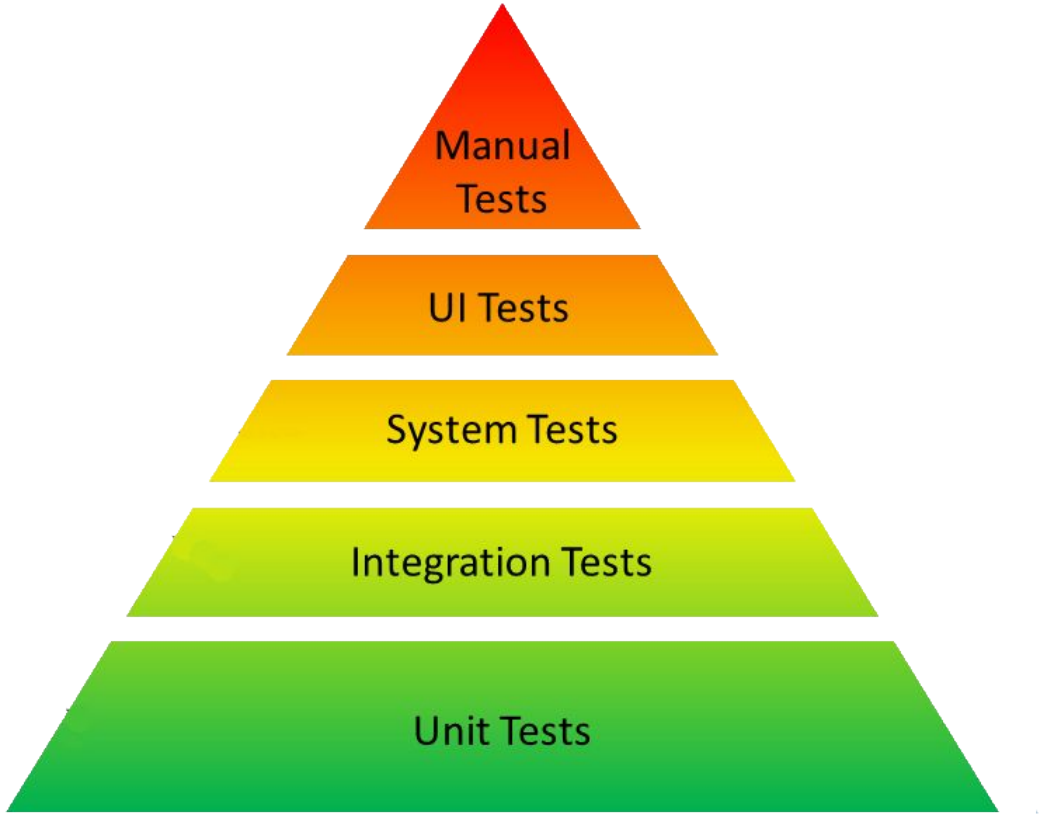
3 - Tests help you PROVE IT WORKS

- Tests that track performance prove you didn't introduce inefficiencies
- Classes with full test compliments can be modified freely
- Integration tests make sure your server doesn't break your client BEFORE customers start using it
- Your boss doesn't understand code (he's too important)
  - Give him the test results output and go home for the day

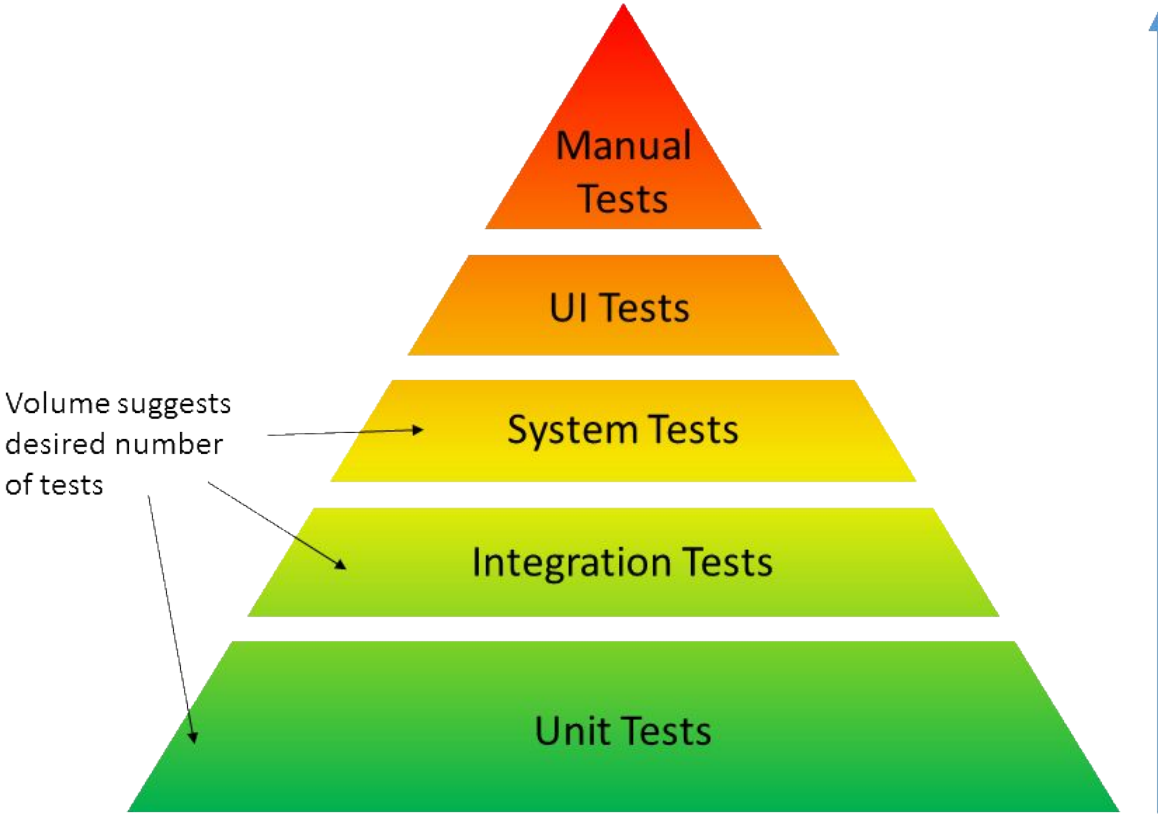
4 - You'll Wish you had

- Regret about test coverage is the #1 tech debt on every team I've ever been on

Types of Testing



Types of Testing

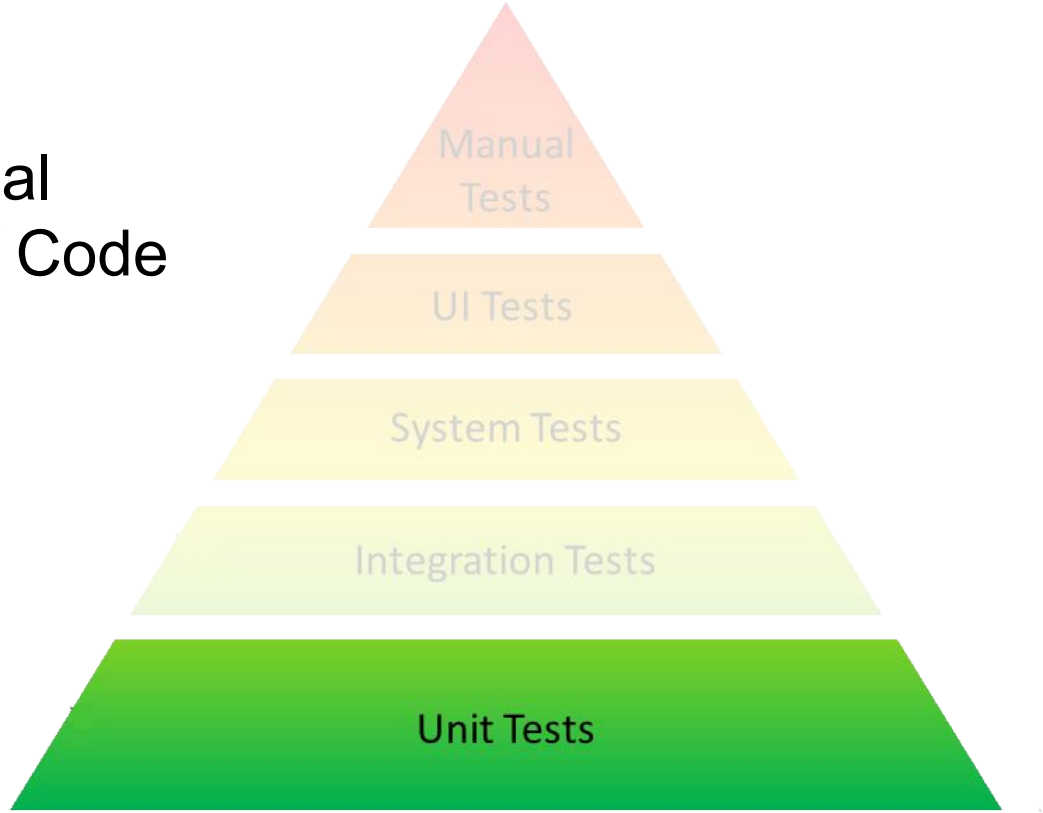


Rising with the pyramid:

- Complexity
- Fragility
- Cost of maintenance
- Execution time
- Time to locate bug on test failure

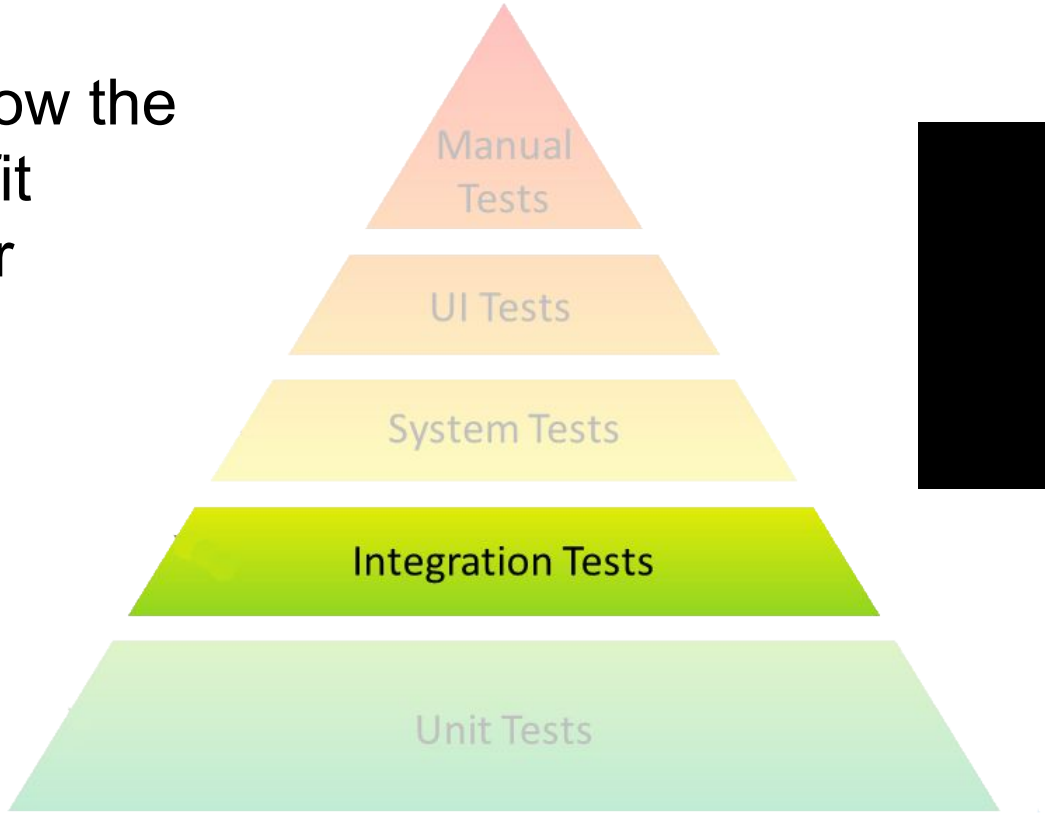
Types of Testing

Tests
Individual
Units of Code



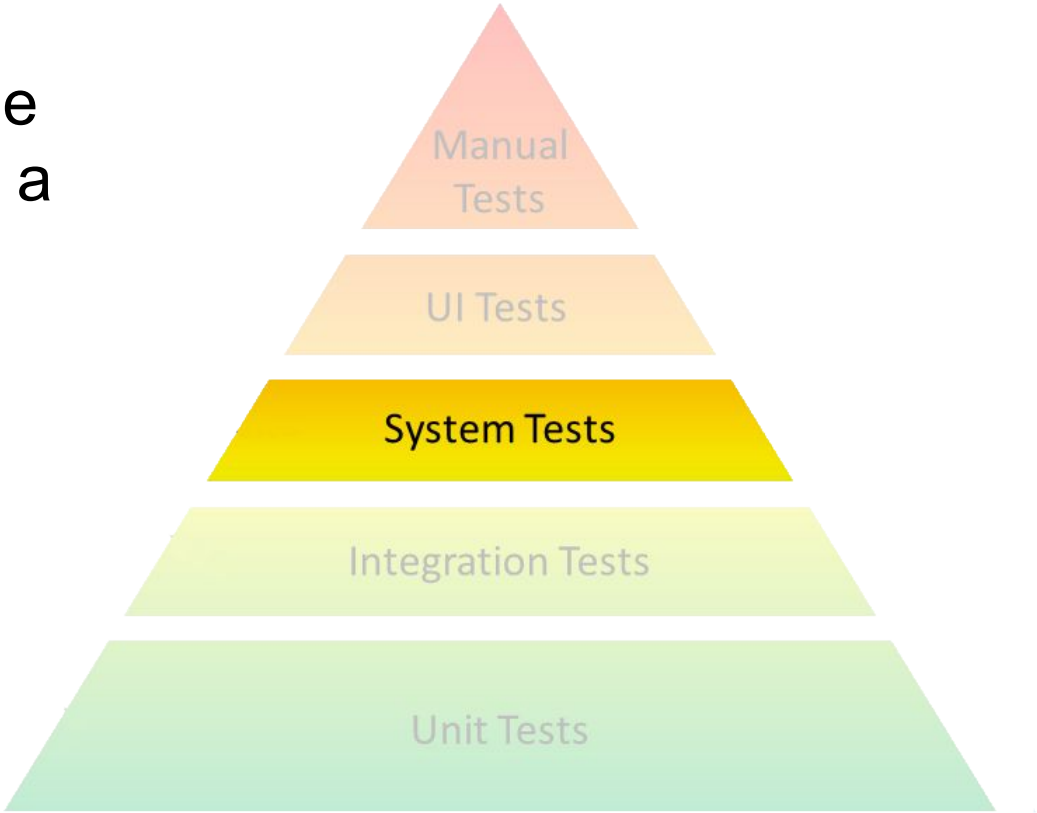
Types of Testing

Tests how the pieces fit together



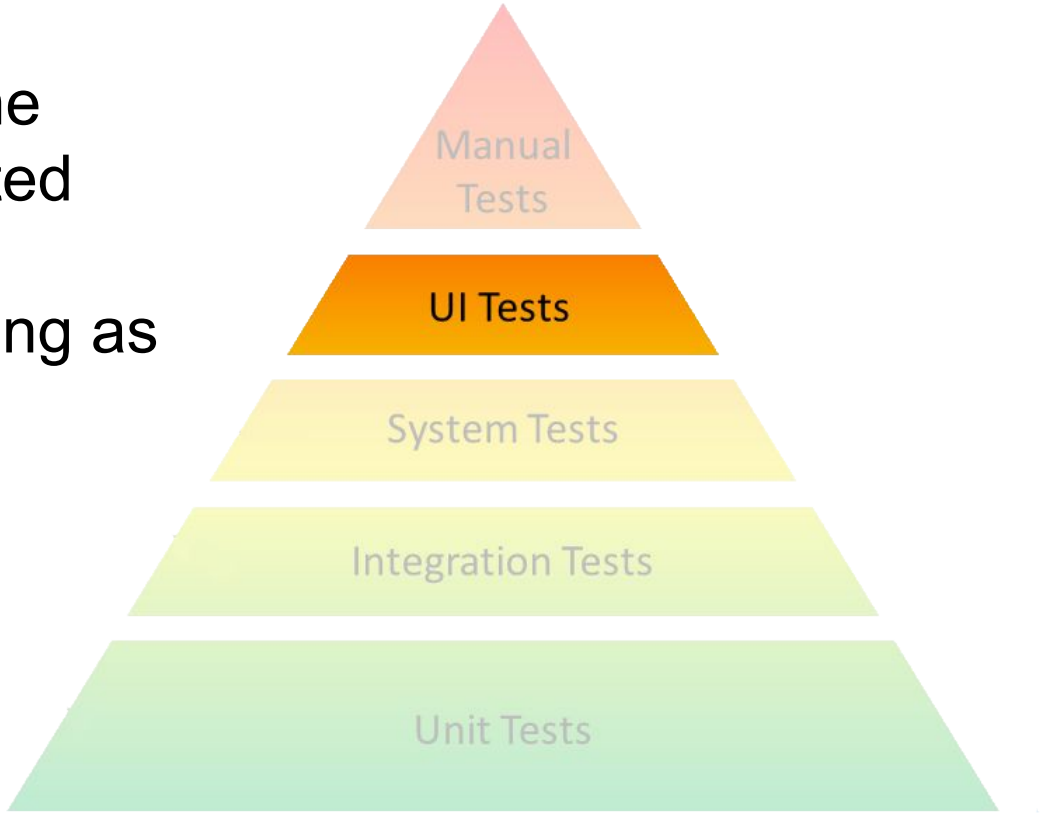
Types of Testing

Tests the thing as a whole



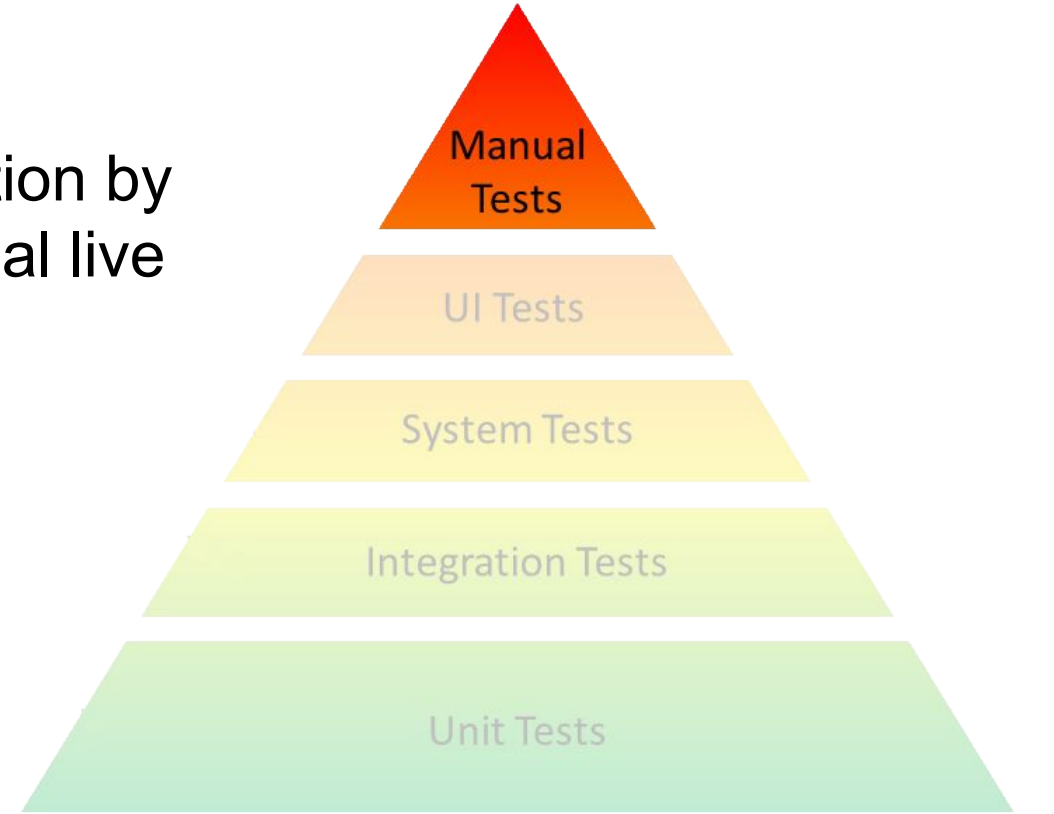
Types of Testing

Tests the completed app by interacting as a user



Types of Testing

Manual interaction by an actual live person



The People

Everyone here does a bit of everything

Software Developers

Manual Testers

Automation Engineers & SDETs

Software Developers

- Write the code
- Write most of the unit tests
- Write some UI Tests
- Have their hands in everything

Manual Testers

- Heavily involved in the design process
- Tests by manual interaction with the product
- A Software Developer's best friend
- Interacts strongly with support team
  - (reproduce customer issues, verify fixes, etc...)
- Verifies software as "ready to release"

Automation Engineers & SDET

- Write tests that run without human interaction
  - (in an ideal world)
- Work closely with manual testers
- Design tests to alleviate the burden of repetitive tasks
- Design tests at scale
- Write tests that verify performance
- SDET: Developer whose product is your product's quality

What do all these people do?

	Write Product Code	Write Unit Tests	Deep Inside Knowledge	Write Automated Tests	Write System & Acceptance Tests	Write Manual Tests	Run Manual Tests & Acceptance Tests
Software Developer	✓	✓	✓	✓	✓	✓	✓
Manual QA Engineer					✓	✓	✓
Automation Engineer				✓	✓	✓	✓
SDET		✓	✓	✓	✓	✓	✓

What tools do we have in Mobile Land?

Both

- Appium
  - Cross platform
  - Single scripting language
  - No internal access
  - Separate scripts per OS

iOS

- XCTest
  - Internal access
  - Native language
- UITest
  - No internal access
  - Native language
  - Auto-record

Android

- JUnit
  - Internal access only
  - Native language
- UIAutomator
  - No internal access
  - Auto-record
- Espresso
  - Native language
  - Auto-record

Testing with Internal Knowledge

Both

- Appium
  - Cross platform
  - Single scripting language
  - No internal access
  - Separate scripts per OS

iOS

- XCTest
  - Internal access
  - Native language
- UITest
  - No internal access
  - Native language
  - Auto-record

Android

- JUnit
  - Internal access only
  - Native language
- UIAutomator
  - No internal access
  - Cross Application
- Espresso
  - Native language
  - Auto-record

Testing from the Outside

Both

- Appium
  - Cross platform
  - Single scripting language
  - No internal access
  - Separate scripts per OS

iOS

- XCTest
  - Internal access
  - Native language
- UITest
  - No internal access
  - Native language
  - Auto-record

Android

- JUnit
  - Internal access only
  - Native language
- UIAutomator
  - No internal access
  - Cross Application
- Espresso
  - Native language
  - Auto-record

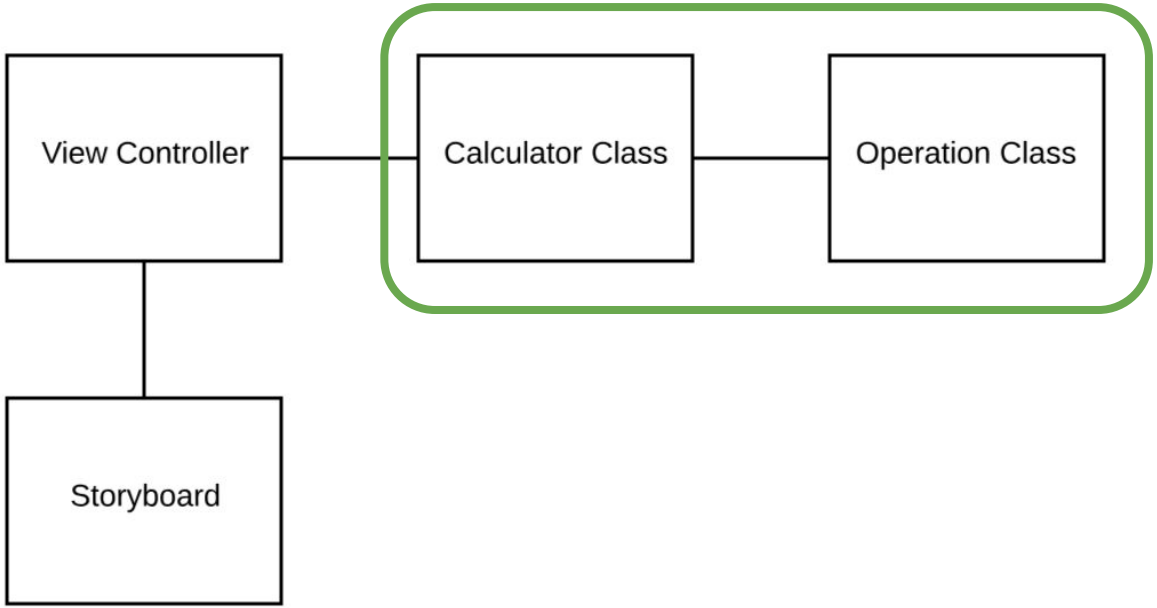
Let’s look at an example App

Dig in with some concrete examples

FIRST: what does the app do?
Hey Sean… Run the thing now

Separation of functionality

Encapsulation helps with testing



The meat of the implementation

```
class Calculator {

    // This performs an operation synchronously
    func performCalculation( input:String? ) -> String? {
      // convert the string to an Int
      if let input = input, let intInput = Int(input) {
        // return the output of the operation
        return String(SquareOperation().operate(input: intInput))
      }
      // if anything failed, return nil
      return nil
    }

    ...
```

Deciding what to test

This is a tricky topic

- What does it do?
  - What does it operate on?
  - What range do its inputs fall under?
  - What are its outputs?

- What doesn't it do?
  - What are the error conditions that cause it to fail?
  - How does the caller/user know that it failed?

Testing the specific “What does it do?”

```
func testWhatItDoes() {
  let c = Calculator()
  let returnValue = c.performCalculation(input: "10")

  // It should give us back something that is non-nil
  guard let resultString = returnValue else {
    XCTFail("Valid input should result in a non-nil response")
    return
  }

  // It should give us back something that is an integer
  guard let resultInt = Int(resultString) else {
    XCTFail("All valid responses should be Integers")
    return
  }

  // It should give us a specific result of 100
  XCTAssert(resultInt==100,"Failed to calculate the correct result for 10.
      Expected: 100 Actual: \(resultInt) ")
}
```

Make the test more generic

```
private func testResultValue( input: Int, result: Int) {
  let c = Calculator()
  let returnValue = c.performCalculation(input: String(input))

  guard let resultString = returnValue else {
    XCTFail("Valid input should result in a non-nil response")
    return
  }

  guard let resultInt = Int(resultString) else {
    XCTFail("All valid responses should be Integers")
    return
  }

  XCTAssert(resultInt==result,"Failed to calculate the correct result for \(input).  Expected: \(r
      Actual: \(resultInt) ")
}
```

Test the oddball values and extents

```
func testResults() {
  testResultValue(input: 10,result: 100)
  testResultValue(input: 1,result: 1)
  testResultValue(input: 0,result: 0)
  testResultValue(input: 50,result: 2500)
}
```

What doesn't it do?

```
func testInvalidInput() {
  let c = Calculator()
  let result = c.performCalculation(input: "Frodo Baggins")
  XCTAssert(result==nil,"invalid input should result in a nil response")
}
```

We didn’t think of everything

- There is a ton more stuff we could be validating
  - Out of range errors on Integers
  - Floating Point Numbers
  - Unintentional conversions of things like “1 is great” to the number 1

- How does one decide what is important?

Deciding what to test

Can anything help?

Test Driven Development

Any guesses what this is?

Test Driven Development

Methodology where we WRITE THE TESTS FIRST!

Tough concept to implement (few places I've been enforce it)

Flip the order of your process. This is not always possible, but when it is… it really works

Isolate parts and encapsulate functionality first. Then write tests for the individual pieces.

Testing Techniques

Once you know the “what” you then need to know the “how”

- Mocking & Dependency Injection
- Asynchronous Tests
- UI Testing

Dependency Injection & Mocking

These go hand in hand

WHAT DO YOU THINK THESE ARE?

Dependency Injection & Mocking

Testing in isolation

- Mocking
  - Make fake representations of objects and classes
  - Allows you to control behavior
  - Simulate hard to reproduce situations

Dependency Injection & Mocking

Testing in isolation

- Mocking
  - Make fake representations of objects and classes
  - Allows you to control behavior
  - Simulate hard to reproduce situations

- Dependency Injection
  - Complicated name for a simple concept
  - Allow Objects to receive their dependencies instead of specifying them
  - Giving an object its instance variables (duh)

Dependency Injection Example

```
// DEPENDANCY INJECTION!!!
var operation: Operation = SquareOperation()

// This performs an operation synchronously
func performCalculation( input:String? ) -> String? {
  // convert the string to an Int
  if let input = input, let intInput = Int(input) {
    // return the output of the operation
    return String(operation.operate(input: intInput))

  }
  // if anything failed, return nil
  return nil
}
```

Dependency design

```
// This interface isolates the dependency
protocol Operation {
  func operate(input:Int) -> Int
}

// This is the actual implementation of the dependency
class SquareOperation: Operation {
  func operate(input:Int) -> Int {
    return input * input
  }
}
```

Mocking the Dependency

```
// This is a mocked implementation of the dependency
class MockOperation: Operation {
  func operate(input:Int) -> Int {
    if input == 20 {
      return 20
    }
    else {
      return 100
    }
  }
}
```

A Test Using Dependency Injection

```
func testDependency() {
  let c = Calculator()
  c.operation = MockOperation()

  let result = c.performCalculation(input: "20")
  XCTAssert(result=="20","Failed to calculate the mocked result for 20.
      Expected: 20 Actual: \(result ?? "") ")

  let result2 = c.performCalculation(input: "0")
  XCTAssert(result2=="100","Failed to calculate the mocked result for 0.
      Expected: 100 Actual: \(result ?? "") ")
}
```

Asynchronous Testing

The asynchronous code we want to test

```
// This performs an operation asynchronously
func performAsynchronousCalculation( input:String?, operation:Operation,
    callback: @escaping (String?)->() ) {
  DispatchQueue.main.asyncAfter(deadline: .now() + 2.0) {
    if let answer = self.performCalculation(input:input) {
      callback("Async: \(answer)")
    }
    else {
      callback(nil)
    }
  }
}
```

Asynchronous Testing

Using Expectations

```
func testAsynchronousResult() {

  let verifyNil = expectation(description: "Nil value received")

  Calculator().performAsynchronousCalculation(input: "Bilbo Baggins")
      { (answer) in
    if answer == nil {
      verifyNil.fulfill()
    }
  }

  waitForExpectations(timeout: 5.0) { (error) in
    if let error = error {
      XCTFail(error.localizedDescription)
    }
  }

}
```

Asynchronous Testing

Using Inverted Expectations

```
func testAsyncWithInverted() {

  let gotNil = expectation(description: "Nil value received")
  gotNil.isInverted = true

  let valueCorrect = expectation(description: "Appropriate Value received")

  Calculator().performAsynchronousCalculation(input: "10") { (answer) in
    guard let nonNilAnswer = answer else {
      gotNil.fulfill()
      return
    }

    if nonNilAnswer == "Async: 100" {
      valueCorrect.fulfill()
    }
  }

  waitForExpectations(timeout: 5.0) { (error) in
    if let error = error {
      XCTFail(error.localizedDescription)
    }
  }

}
```

A Real World Example

Real mechanism for validating a complex manager object

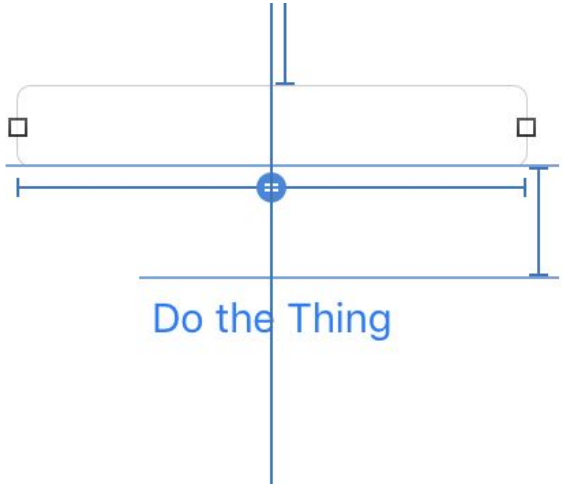
UI Testing

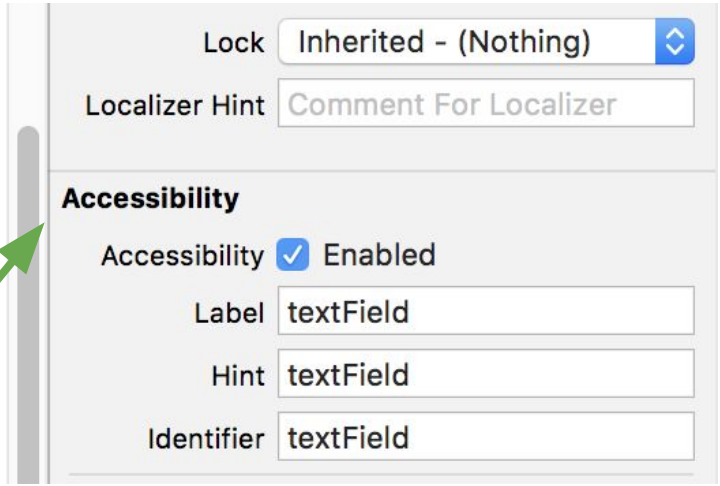
- Tests the app from the outside by performing actions on the UI
- Validates the state of the application under various conditions
- It IS possible to mock pieces of the application during UI tests
  - but it is VERY hacky
  - Sure… we do it. Sometimes you gotta break the rules

UI Testing

How do I get started?

- You must set up controls you wish to use properly
  - fill in Accessibility settings in Interface Builder

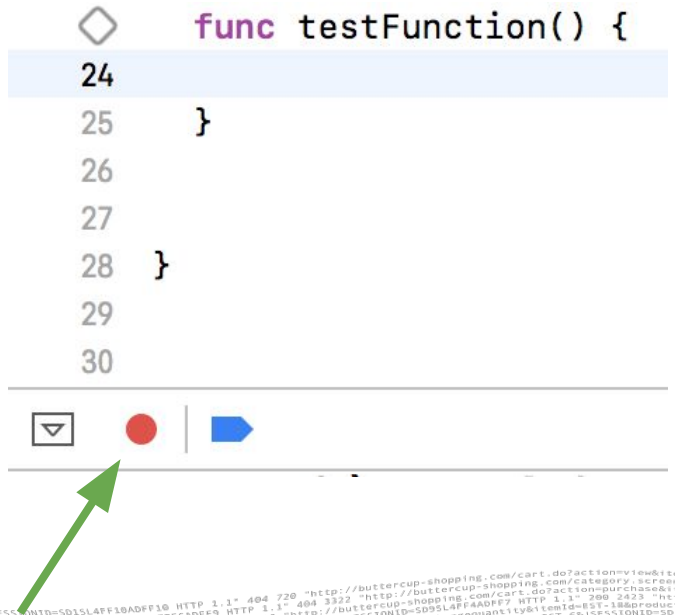




UI Testing

How do I get started?

- Record by placing your cursor inside the UI Test and tapping the red record button



UI Testing

What to watch out for

- Labels are funny
  - Accessibility setting should NOT be filled in if you want to check label values dynamically
- This whole thing is flaky
  - Expect XCode to barf on you 2 to 3 times an hour
  - But, once the tests are written they execute well and consistently
- The recording is ONLY a guide
  - You will need to hand edit the code EVERY time

UI Testing

A very simple UITest for our App

```
func testFunction() {
  let app = XCUIApplication()
  app.textFields["textField"].tap()
  app.textFields["textField"].typeText("10")
  app.buttons["doItButton"].tap()

  XCTAssert(app.staticTexts.element(matching:.any, identifier:
      "answerLabel").label == "100")

  app.textFields["textField"].tap()
  app.textFields["textField"].clearAndEnterText(text: "5")
  app.buttons["doItButton"].tap()

  XCTAssert(app.staticTexts.element(matching:.any, identifier:
      "answerLabel").label == "25")
}
```

A Real World Example

Real UI Test

Further Research

Some useful topics if you want to go further

- iOS Network Testing
  - URLProtocol on iOS
    - https://kandelvijaya.com/2017/04/30/urlprotocolandunittesting/
  - Performance Testing in iOS (self.measure)
- Android Espresso Test Recorder
  - https://developer.android.com/studio/test/espresso-test-recorder
- Android UIAutomator
  - https://developer.android.com/training/testing/ui-automator