# Level up your ~~unit~~ testing approach using BDD-style tests

Starring Quick and Nimble

**Presented By:** Jeff Roberts
**Email:** jeff@brax.tv
**Twitter:** @JeffBNimble
**GitHub:** JeffBNimble

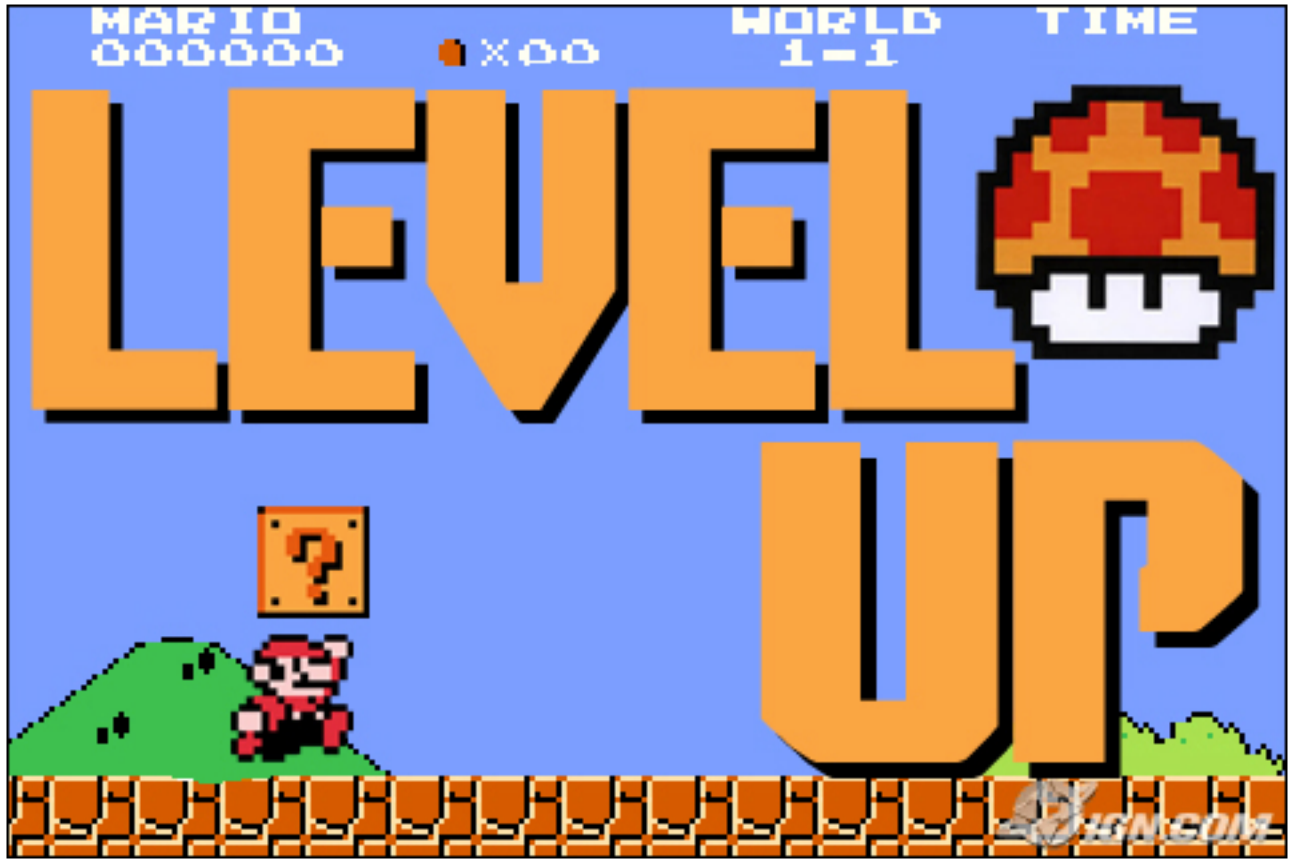
# About Me

- I'm old (compared to you)
- Coding forever (~30 years)
- Professionally since 1986
- Consulting since 1991
- Riot since 2008 (we're hiring)
- Languages
  - RPG II, III, 400 [1986]
  - Smalltalk [1992]
  - Java [1996]
  - HTML/Javascript/CSS [2002]
  - Flex/AIR [2007]
  - iOS/Android [2012]
  - C++/Go [2015/2016]
- ~ 200 tests in the past month
- Sharing
  - brax.tv

DEVELOPER • TUTORIALS
BRAX
- Brain Wax -

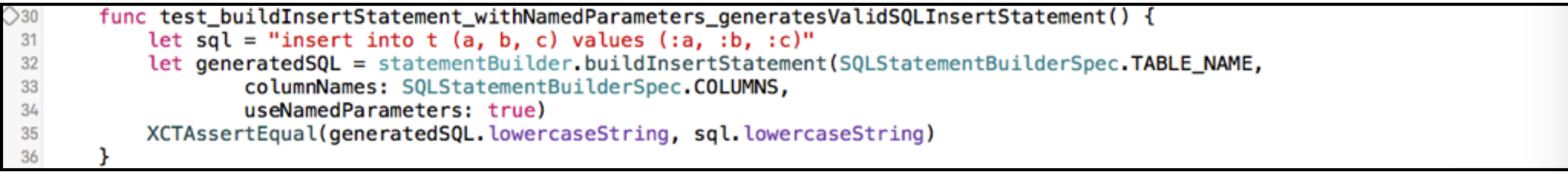
# Agenda

Unit vs. Integration vs. Functional

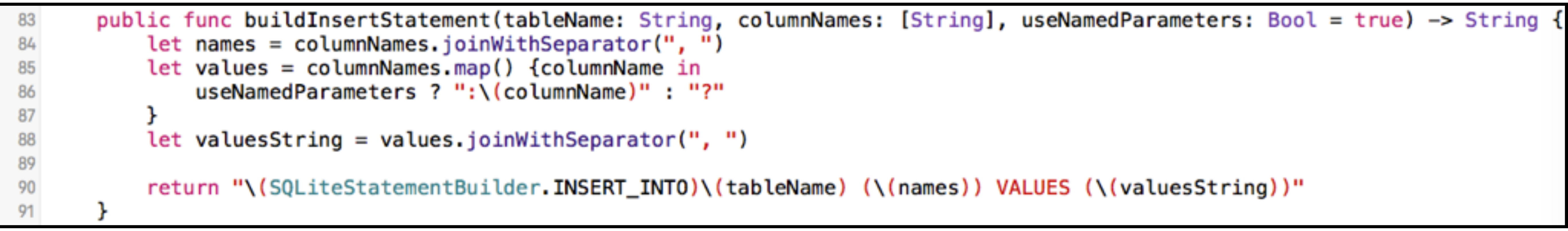


## XCTest Review

# XCTest Review



```
    func test_buildInsertStatement_withNamedParameters_generatesValidSQLInsertStatement() {
        let sql = "insert into t (a, b, c) values (:a, :b, :c)"
        let generatedSQL = statementBuilder.buildInsertStatement(SQLStatementBuilderSpec.TABLE_NAME,
                columnNames: SQLStatementBuilderSpec.COLUMNS,
                useNamedParameters: true)
        XCTAssertEqual(generatedSQL.lowercaseString, sql.lowercaseString)
    }
```



```
    public func buildInsertStatement(tableName: String, columnNames: [String], useNamedParameters: Bool = true) -> String {
        let names = columnNames.joinWithSeparator(", ")
        let values = columnNames.map() {columnName in
            useNamedParameters ? ":\(columnName)" : "?"
        }
        let valuesString = values.joinWithSeparator(", ")

        return "\(SQLiteStatementBuilder.INSERT_INTO)\(tableName) (\(names)) VALUES (\(valuesString))"
    }
```

✓ ✗

# XCTest Anatomy

```swift
import XCTest

class SQLStatementBuilderTest : XCTestCase {
 var statementBuilder : SQLiteStatementBuilder
 override func setUp() {
   super.setUp()

   statementBuilder = SQLiteStatementBuilder()
 }

 override func tearDown() {
   super.tearDown()
 }

 func test_buildInsertStatement_withNamedParameters_generatesValidSQLInsertStatement() {
     let sql = "insert into t (a, b, c) values (:a, :b, :c)"
     let generatedSQL = statementBuilder.
       buildInsertStatement(SQLStatementBuilderSpec.TABLE_NAME,
         columnNames: SQLStatementBuilderSpec.COLUMNS,
         useNamedParameters: true)
     XCTAssertEqual(generatedSQL.lowercaseString, sql.lowercaseString)
 }

}
```

# XCTest Ancestry

Free Pascal (FPCUnit)

Fortran (fUnit, pFUnit)

Delphi (DUnit)

**Java (JUnit)**

Haskell (HUnit)

Eiffel (Auto-Test)

PHP (PHPUnit)

Erlang (EUnit)

JavaScript (JSUnit)

Coldfusion (MXUnit)

Microsoft .NET (NUnit)

OCaml (OUnit)

C++ (CPPUnit, CxxTest)

Python (PyUnit)

C# (NUnit)

Qt (QTestLib)

C (CUnit)

Perl (Test::Class and Test::Unit)

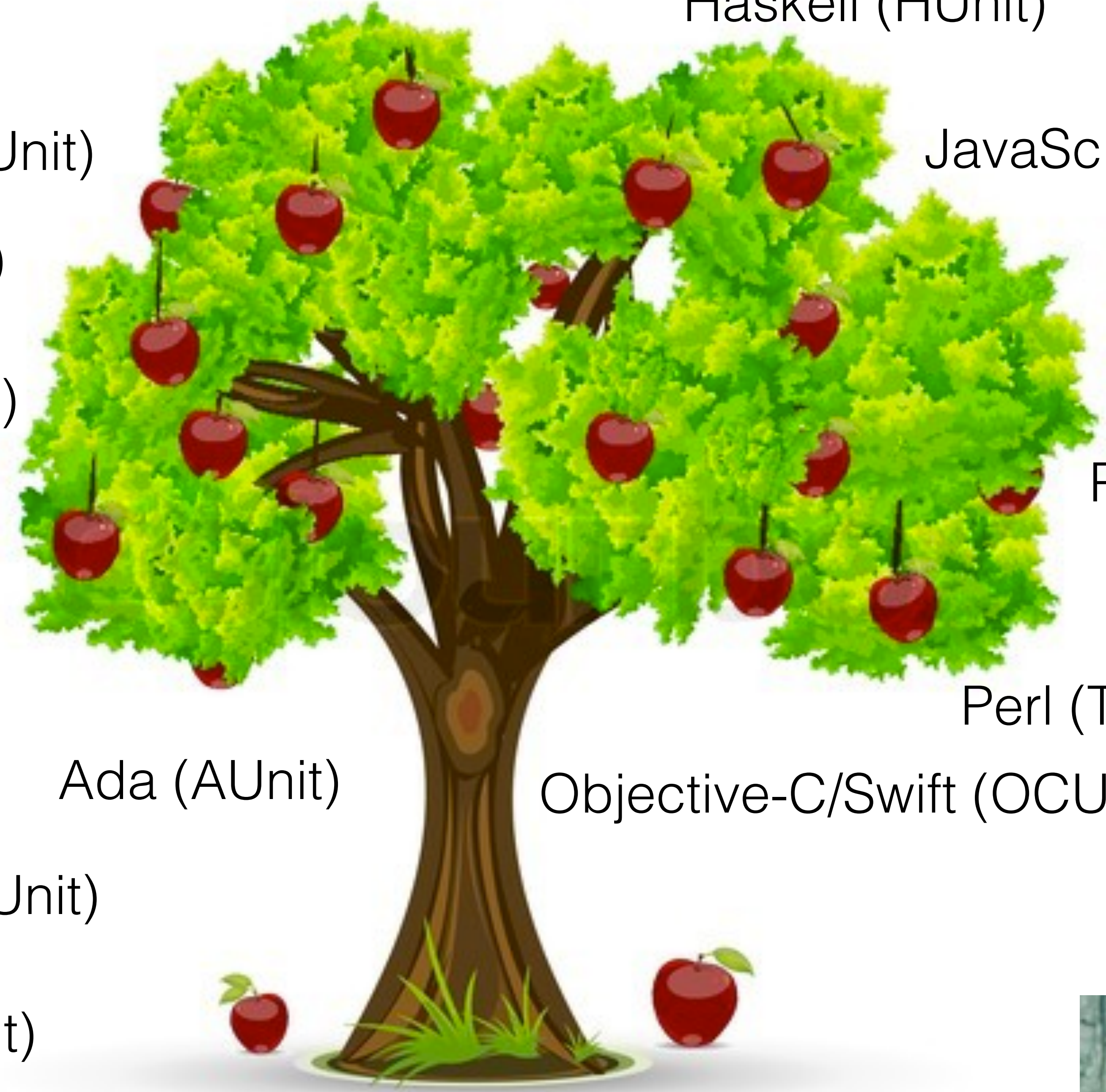
Ada (AUnit)

Objective-C/Swift (OCUnit, **XCTest**)

Actionscript (FlexUnit)

Ruby (Test::Unit)

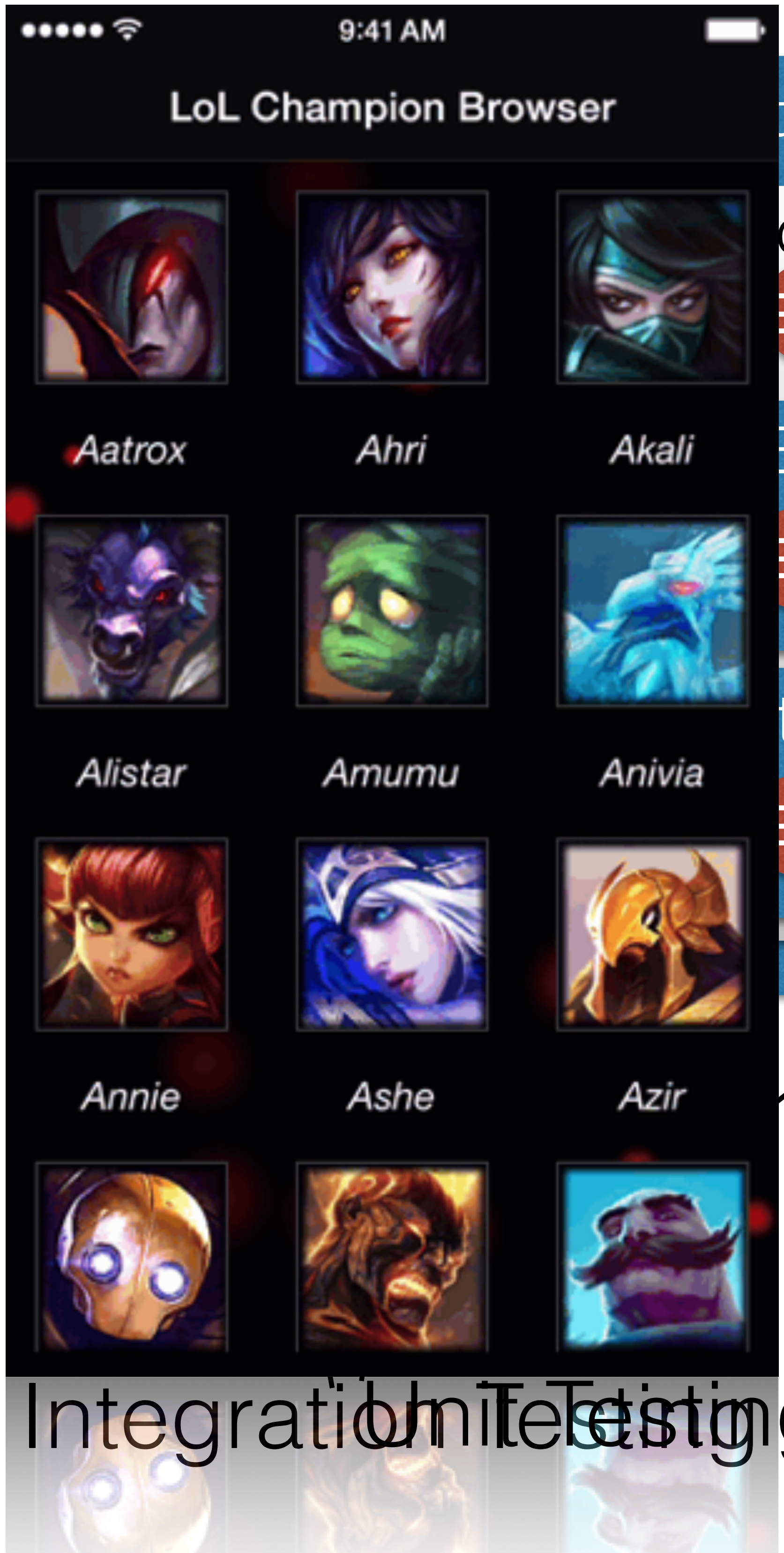
R (RUnit)

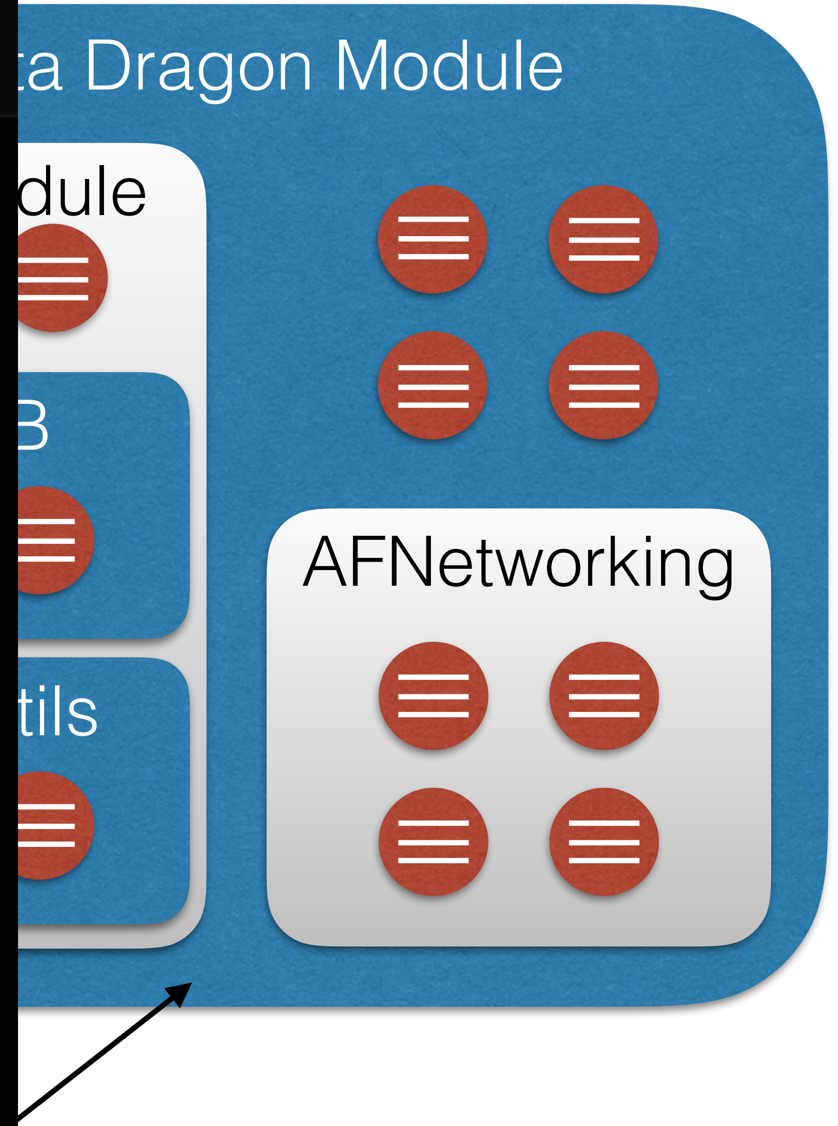


1994 - SUnit by Kent Beck

# Types of Testing

Functional Testing





Integration Testing

Unit Testing

Riot API

# Unit Test: Class Under Test

```
public protocol SQLStatementBuilder {
    func buildDeleteStatement(
     tableName:String,
     selection:String?) -> String

    func buildInsertStatement(
     tableName:String,
     columnNames:[String],
     useNamedParameters:Bool) -> String

    func buildSelectStatement(
     tableName:String,
     projection:[String]?,
     selection:String?,
     groupBy:String?,
     having:String?,
     sort:String?) -> String

   func buildUpdateStatement(
     tableName:String,
     updatingColumnNames:[String],
     selection:String?,
     useNamedParameters:Bool) -> String
}

public class SQLiteStatementBuilder : SQLStatementBuilder {
  . . .
}
```

# XCTest: Unit Test

```
import XCTest

class SQLStatementBuilderTest : XCTestCase {
    static let TABLE_NAME = "t"
    static let COLUMN_A = "a"
    static let COLUMN_B = "b"
    static let COLUMN_C = "c"
    static let COLUMNS = [COLUMN_A, COLUMN_B, COLUMN_C]

    var statementBuilder : SQLStatementBuilder!

    override func setUp() {
        super.setUp()

        statementBuilder = SQLiteStatementBuilder()
    }

    // SQL INSERT Tests
    func test_buildInsertStatement_withNamedParameters_generatesValidSQLInsertStatement() {
        let sql = "insert into t (a, b, c) values (:a, :b, :c)"
        let generatedSQL = statementBuilder.buildInsertStatement(
                SQLStatementBuilderSpec.TABLE_NAME,
                columnNames: SQLStatementBuilderSpec.COLUMNS,
                useNamedParameters: true)
        XCTAssertEqual(generatedSQL.lowercaseString, sql.lowercaseString)
    }
}
```

# XCTest: Unit Test

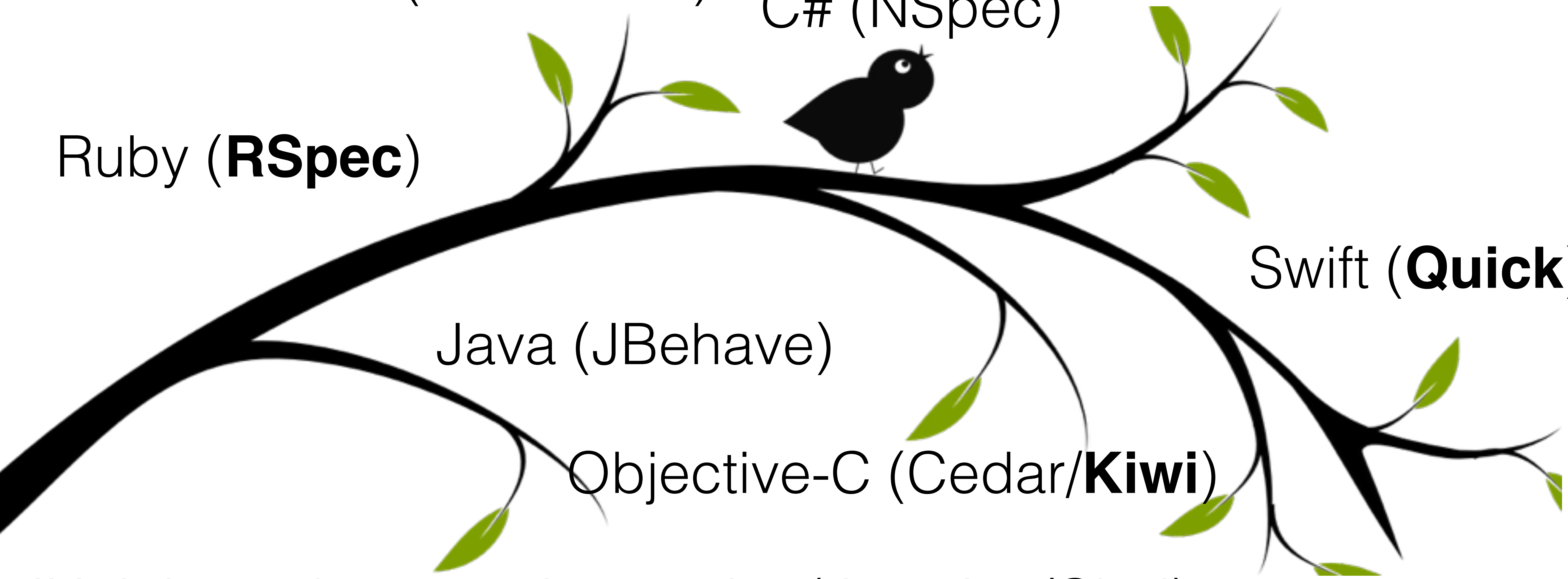
```
import XCTest

class SQLStatementBuilderTest : XCTestCase {

    // SQL INSERT Tests
    func test_buildInsertStatement_withNamedParameters_generatesValidSQLInsertStatement() {
        let sql = "insert into t (a, b, c) values (:a, :b, :c)"
        let generatedSQL = statementBuilder.buildInsertStatement(
                SQLStatementBuilderSpec.TABLE_NAME,
                columnNames: SQLStatementBuilderSpec.COLUMNS,
                useNamedParameters: true)
        XCTAssertEqual(generatedSQL.lowercaseString, sql.lowercaseString)
    }

    func test_buildInsertStatement_withParameterMarkers_generatesValidSQLInsertStatement() {
        let sql = "insert into t (a, b, c) values (?, ?, ?)"
        let generatedSQL = statementBuilder.buildInsertStatement(
                SQLStatementBuilderSpec.TABLE_NAME,
                columnNames: SQLStatementBuilderSpec.COLUMNS,
                useNamedParameters: false)
        XCTAssertEqual(generatedSQL.lowercaseString, sql.lowercaseString)
    }
}
```

# Introducing BDD

- Behavior-Driven Development

  The focus is on writing **specs** instead of tests and **describing** what you are testing by defining the scenarios as **contexts** using sentences, thereby significantly increasing the clarity of your tests.

# BDD Ancestry



2006 - JBehave by Dan North

# Unit Test: Class Under Test

```
public protocol SQLStatementBuilder {
    func buildDeleteStatement(
     tableName:String,
     selection:String?) -> String

    func buildInsertStatement(
     tableName:String,
     columnNames:[String],
     useNamedParameters:Bool) -> String

    func buildSelectStatement(
     tableName:String,
     projection:[String]?,
     selection:String?,
     groupBy:String?,
     having:String?,
     sort:String?) -> String

   func buildUpdateStatement(
     tableName:String,
     updatingColumnNames:[String],
     selection:String?,
     useNamedParameters:Bool) -> String
}

public class SQLiteStatementBuilder : SQLStatementBuilder {
  . . .
}
```

# Building a Spec

```swift
import Quick
import Nimble

class SQLStatementBuilderPendingSpec : QuickSpec {
    override func spec() {
        describe("Given a SQLStatementBuilder") {

            // SQL INSERT
            context("when building an INSERT statement") {

                context("with named parameters") {

                    pending(“then it should generate a valid SQL INSERT statement") {}

                }

                context("with parameter markers") {

                    pending(“then it should generate a valid SQL INSERT statement") {}

                }
            }
        }
    }
}
```

# Pending Spec: Output

```
Pending: then it should generate a valid SQL INSERT statement
Pending: then it should generate a valid SQL INSERT statement
Pending: then it should generate a valid SQL DELETE statement without a WHERE clause
Pending: then it should generate a valid SQL DELETE statement with a WHERE clause
Pending: then it should generate a valid SQL UPDATE statement
Pending: then it should generate a valid SQL UPDATE statement
Pending: then it should generate '*' for the projection
Pending: then it should generate a selection clause for each column in the projection
Pending: then it should not generate a WHERE clause
Pending: then it should generate a valid WHERE clause
Pending: then it should not generate a GROUP BY clause
Pending: then it should generate a valid GROUP BY clause
Pending: then it should not generate a HAVING clause
Pending: then it should generate a valid HAVING clause
Pending: then it should not generate an ORDER BY clause
Pending: then it should generate a valid ORDER BY clause
```

# Converting Pending

```
class SQLStatementBuilderSpec : QuickSpec {
    static let TABLE_NAME = "t"
    static let COLUMN_A = "a"
    static let COLUMN_B = "b"
    static let COLUMN_C = "c"
    static let COLUMNS = [COLUMN_A, COLUMN_B, COLUMN_C]

    override func spec() {
        describe("Given a SQLStatementBuilder") {
            var statementBuilder : SQLStatementBuilder!

            beforeEach {
                statementBuilder = SQLiteStatementBuilder()
            }

            // SQL INSERT
            context("when building an INSERT statement") {

                context("with named parameters") {

                  it("then it should generate a valid SQL INSERT statement") {
                      let sql = "insert into t (a, b, c) values (:a, :b, :c)"
                      let generatedSQL = statementBuilder.buildInsertStatement(
                                SQLStatementBuilderSpec.TABLE_NAME,
                                columnNames: SQLStatementBuilderSpec.COLUMNS,
                                useNamedParameters: true)
                      expect(generatedSQL.lowercaseString).to(equal(sql.lowercaseString))
                  }
```

# Test Observer

- Test output is readable, obvious

```
Given a SQLStatementBuilder
     when building an INSERT statement
        with named parameters
            then it should generate a valid SQL INSERT statement [PASSED]
        with parameter markers
            then it should generate a valid SQL INSERT statement [PASSED]
     when building an UPDATE statement
        with named parameters
            then it should generate a valid SQL UPDATE statement [PASSED]
        with parameter markers
            then it should generate a valid SQL UPDATE statement [PASSED]
     when building a DELETE statement
        without a selection specified
            then it should generate a valid SQL DELETE statement without a WHERE clause [PASSED]
        with a selection specified
            then it should generate a valid SQL DELETE statement with a WHERE clause [PASSED]

    . . .


16 tests run in 261.422991752625ms
16 pass/0 fail
```

# Focused/Excluded

- describe/fdescribe/xdescribe
- context/fcontext/xcontext
- it/fit/xit

# XCTest assertions

- XCTAssertEqual(actual, expected, "expected actual to equal expected")
- XCTAssertTrue(actual, "expected actual to be true")
- XCTAssertFalse(actual, "expected actual to be false”)
- XCTAssertNil(actual, “expected actual to be nil”)
- XCTAssertNotNil(actual, “expected actual to not be nil”)
- XCTAssertThrows(doSomething(), “expected doSomething() to throw”)
- XCTFail(“Fail the test”)

# Leveling Up Assertions

- expect(actual).to(equal(expected))
- expect(actual).toNot(equal(expected))
- expect(actual).notTo(equal(expected))
- expect(actual).toEventually(equal(expected))
- expect(actual).to(beTrue())
- expect(actual).to(beFalse())
- expect(actual).to(beNil())
- expect(actual).notTo(beNil())

## Integration Tests

- Can use BDD approach with Integration tests

# Functional Tests

```swift
public override func spec() {
   describe("Given that I am using the LoL Book of Champions app") {

       context("when I am on the champion browser screen") {

           it("then I can scroll the list of champions until I see Ziggs") {}

           it("then I can tap on Annie and see Annie's skins") {}
       }

       context("when I am on the champion skins browser screen") {

           it("then I can scroll through Annie's skins") {}

           it("then I can tap the back button to go back to the Champions Browser") {}
       }

   }
}
```

# Functional Test Observer

- Test output is readable, obvious

```
Given that I am using the LoL Book of Champions app
 when I am on the champion browser screen
   then I can scroll the list of champions until I see Ziggs [PASSED]
   then I can tap on Annie and see Annie's skins [PASSED]
 when I am on the champion skins browser screen
   then I can scroll through Annie's skins [PASSED]
   then I can tap the back button to go back to the Champions Browser [PASSED]


4 tests run in 58.7978560328484s
4 pass/0 fail
```

# SUMMARY

- Quick tests are XCTests
- Unit, integration and functional
- Switch focus to behavior
- Tests are easier to read, easier to write
- Makes writing tests…..fun?

# References

- **Quick**: https://github.com/Quick/Quick
- **Nimble**: https://github.com/Quick/Nimble
- **My repo's**: https://github.com/JeffBNimble
- **The Art of Unit Testing Second Edition**: Manning Publications
- **Dan North BDD**: http://dannorth.net/introducing-bdd/
- **Kent Beck SUnit**: http://www.macqueen.us/smalltalkReport/ST/91_95/SMAL0403.PDF
- **Me**: jeff@brax.tv, @JeffBNimble, https://github.com/JeffBNimble