What is a unit test?

“Unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures are tested to determine if they are fit for use.”

Wikipedia, from a book on tests

Um… what?

What is an app?

An app is a set of behaviours created by programmer and expected by user.

We, programmers, have a limited cognition. As all humans do.

We can’t always ‘load’ all of the code of our app into our memory.

This means that we can, by accident, change the behaviour of the app.

Preserving behaviour of complex systems is hard. In fact, of any system at all.

Enter unit tests.

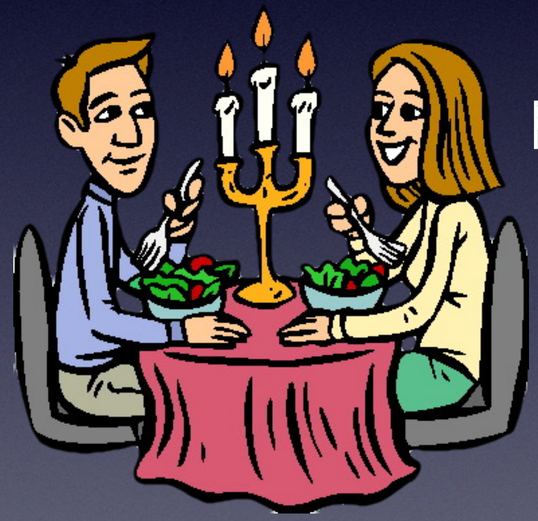
Unit test is a failsafe to make sure app behaviour is preserved.

What is a unit test?

Unit tests test smallest parts of your code in isolation with test code

Unit tests test smallest parts of your code in **isolation** with test code

Test isolation



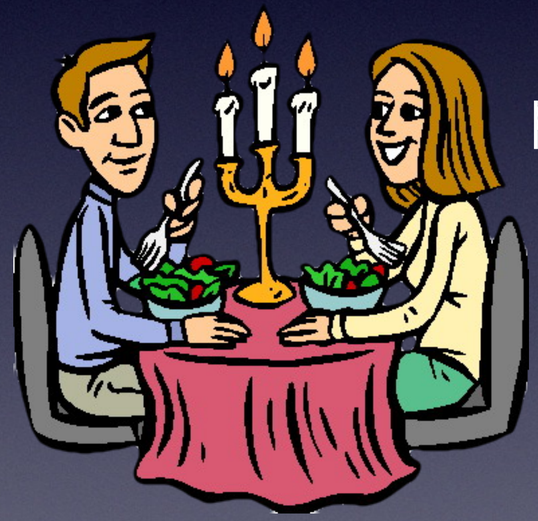
processOrder

getDishes

Table

Waiter

Cook



processOrder

getDishes

I'm
a cook

Fake cook

Table

Waiter

Cook

Why isolate?

Unit test lifecycle

Unit test lifecycle

- Arrange
- Act
- Assert

When a unit test is not a unit test?

When a unit test is not a unit test?

by Michael Feathers

A test is not a unit test if…

- It talks to a database
- It communicates across network
- It touches the file system
- You have to do special things to your environment to run it (edit config files etc)

A test is not a unit test if…

- It talks to a database
- It communicates across network
- It touches the file system
- You have to do special things to your environment to run it (edit config files etc)

A test is not a unit test if…

- It talks to a database
- It communicates across network
- It touches the file system
- You have to do special things to your environment to run it (edit config files etc)

A 100 ms tests is a
very slow test.

A 100 ms tests is a **very** slow test.

Thus touching file system, network, database etc is a no no.

1500 tests each running 100 ms. That’s 150 seconds. Two and a half minutes.

Where does TDD fit in all this?

In TDD you always
write test first. Always.

TDD is not “just adding tests first”. It’s a complete workflow.

TDD is a great way to
“start” when you’re not
in the zone…

… and way to remind yourself what you've been working on yesterday!

TDD is a great way to determine how complex your code has become.

You just have to listen.

Have to fake seven objects to isolate test?

Have to inject a fake
into a fake into a fake?

This always points to an overcomplicated design.

And your tests are here to point that out. Very clearly.

What unit tests can’t do?

Unit tests are never a guarantee that you won’t ship a bug.

But they’re damn good at greatly reducing amount of bugs. And time spent on QA.

Are unit tests an invaluable tool for writing great software? Heck yes. Am I going to produce a poor product if I can't unit test? Hell no.

Jonathan Rasmusson

http://agilewarrior.wordpress.com/2012/10/06/its-not-about-the-unit-tests/

Specta

Specta

BDD Testing Framework

Wait what? BDD?

Wait what? BDD?

Weren’t we supposed to do TDD?

BDD builds upon TDD by formalising the good habits of the best TDD practitioners.

Matt Wynne,
XP Evangelist

http://blog.mattwynne.net/2012/11/20/tdd-vs-bdd/

Good habits

- Work outside-in
- Use examples to clarify requirements
- Use ubiquitous language

Thanks Matt!!

Technical stuff now

Based on XCTest

Minimalistic implementation

Syntax

```
SPEC_BEGIN(Example)

describe(@"Example specs", ^{

});

SPEC_END
```

```
SPEC_BEGIN(Example)

describe(@"Example specs", ^{


});

SPEC_END
```

```
SPEC_BEGIN(Example)

describe(@"Example specs", ^{


});

SPEC_END
```

```
SPEC_BEGIN(Example)

describe(@"Example specs", ^{


});

SPEC_END
```

```
SPEC_BEGIN(Example)

describe(@"Example specs", ^{
    it(@“should check compiler sanity", ^{
        expect(YES).to.beTruthy();
    });
});

SPEC_END
```

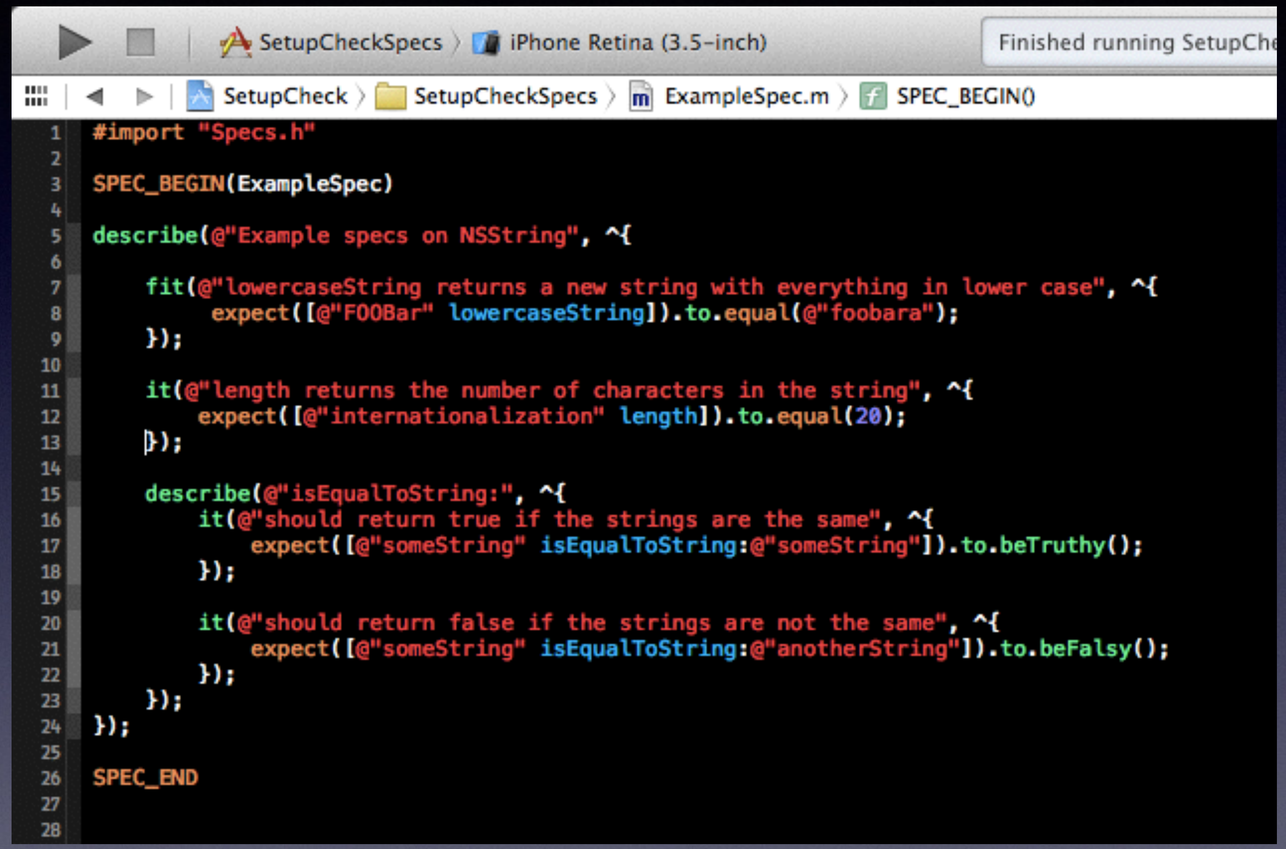
Describe/Context blocks

Used to make tests more readable.

And isolate behaviour for different scenarios.

```
describe(@"NSNumber", ^{

    describe(@"when created with the default constructor", ^{

        it(@“should have 0 as contained int value", ^{
            NSNumber *number = [[NSNumber alloc] init];
            expect([number integerValue]).to.equal(0);
        });
    });

    context(@"when constructed with an int", ^{

        it(@“should have 42 as contained int value", ^{
            NSNumber *number = [[NSNumber alloc] initWithInt:42];
            expect([number integerValue]).to.equal(42);
        });
    });
});
```

You can have as many nested describes as you want.




```
#import "Specs.h"

SPEC_BEGIN(ExampleSpec)

describe(@"Example specs on NSString", ^{

    fit(@"lowercaseString returns a new string with everything in lower case", ^{
        expect([@"FOOBar" lowercaseString]).to.equal(@"foobara");
    });

    it(@"length returns the number of characters in the string", ^{
        expect([@"internationalization" length]).to.equal(20);
    });

    describe(@"isEqualToString:", ^{
        it(@"should return true if the strings are the same", ^{
            expect([@"someString" isEqualToString:@"someString"]).to.beTruthy();
        });

        it(@"should return false if the strings are not the same", ^{
            expect([@"someString" isEqualToString:@"anotherString"]).to.beFalsy();
        });
    });
});

SPEC_END
```

Before/After each blocks

```
beforeEach(^{
    appDelegate = [[AppDelegate alloc] init];
});

afterEach(^{
   appDelegate = nil;
});

it(@"should have a window", ^{
   expect(appDelegate.window).to.beKindOf([UIWindow class]);
});
```

1

```
beforeEach(^{
    appDelegate = [[AppDelegate alloc] init];
});
```

2

```
it(@"should have a window", ^{
    expect(appDelegate.window).to.beKindOf([UIWindow class]);
});
```

3

```
afterEach(^{
    appDelegate = nil;
});
```

Let’s write our very first unit test!

Hands on!

Configuring tests

Focusing tests

Focusing tests

```
fdescribe(@"Example specs on NSString", ^{

fit(@"lowercaseString returns a new string with
everything in lower case", ^{

fcontext(@"init with damping", ^{
```

PENDING

PENDING

```
it(@"lowercaseString returns a new string with
everything in lower case", PENDING);
```

x’ing tests

x’ing tests

```
xdescribe(@"Example specs on NSString", ^{

xit(@"lowercaseString returns a new string with
everything in lower case", ^{

xcontext(@"init with damping", ^{
```

Unit tests results

Unit tests results

How to understand the output?

Xcode, AppCode, Command Line

All give the same results. Devil is in the details

```
(…)

-[SpecSuiteName passing_spec_name]
  Test Case '-[SpecSuiteName passing_spec_name]' started.
  Test Case '-[SpecSuiteName passing_spec_name]' passed
(0.271 seconds).

-[SpecSuiteName failling_spec_name]
  Test Case '-[SpecSuiteName failling_spec_name]' started.
  Test Case '-[SpecSuiteName failling_spec_name]' failed
(0.002 seconds).

(…)

Executed 2 tests, with 1 failure (1 unexpected) in 0.273
(0.278) seconds

2 tests; 0 skipped; 1 failure; 1 exception; 0 pending
```

```
(…)

-[SpecSuiteName passing_spec_name]
  Test Case '-[SpecSuiteName passing_spec_name]' started.
  Test Case '-[SpecSuiteName passing_spec_name]' passed
(0.271 seconds).

-[SpecSuiteName failling_spec_name]
  Test Case '-[SpecSuiteName failling_spec_name]' started.
  Test Case '-[SpecSuiteName failling_spec_name]' failed
(0.002 seconds).

(…)

Executed 2 tests, with 1 failure (1 unexpected) in 0.273
(0.278) seconds

2 tests; 0 skipped; 1 failure; 1 exception; 0 pending
```

```
(…)

-[SpecSuiteName passing_spec_name]
  Test Case '-[SpecSuiteName passing_spec_name]' started.
  Test Case '-[SpecSuiteName passing_spec_name]' passed
(0.271 seconds).

-[SpecSuiteName failling_spec_name]
  Test Case '-[SpecSuiteName failling_spec_name]' started.
  Test Case '-[SpecSuiteName failling_spec_name]' failed
(0.002 seconds).

(…)

Executed 2 tests, with 1 failure (1 unexpected) in 0.273
(0.278) seconds

2 tests; 0 skipped; 1 failure; 1 exception; 0 pending
```

```
(…)

-[SpecSuiteName passing_spec_name]
  Test Case '-[SpecSuiteName passing_spec_name]' started.
  Test Case '-[SpecSuiteName passing_spec_name]' passed
(0.271 seconds).

-[SpecSuiteName failling_spec_name]
  Test Case '-[SpecSuiteName failling_spec_name]' started.
  Test Case '-[SpecSuiteName failling_spec_name]' failed
(0.002 seconds).

(…)

Executed 2 tests, with 1 failure (1 unexpected) in 0.273
(0.278) seconds

2 tests; 0 skipped; 1 failure; 1 exception; 0 pending
```

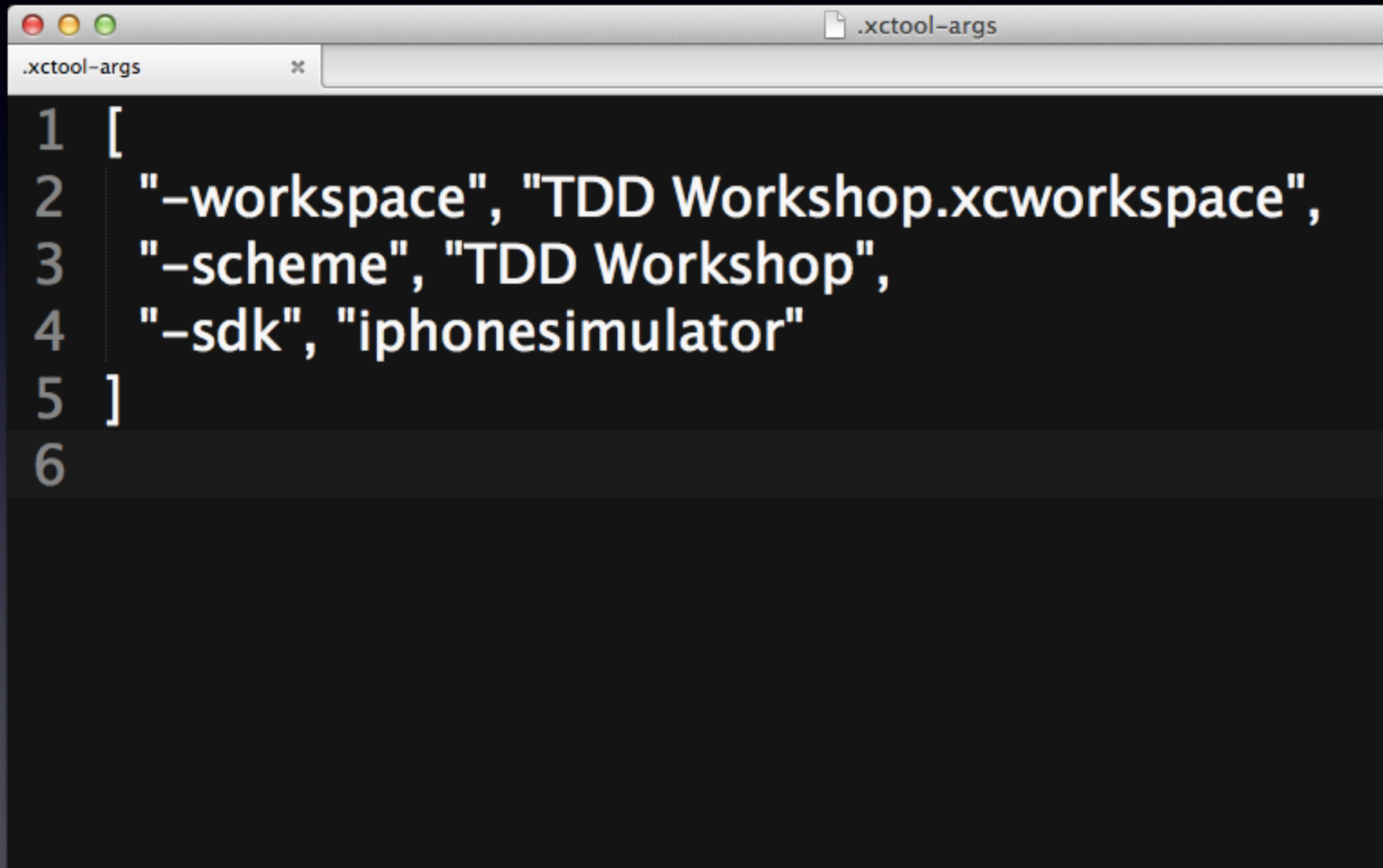
```
(…)

-[SpecSuiteName passing_spec_name]
  Test Case '-[SpecSuiteName passing_spec_name]' started.
  Test Case '-[SpecSuiteName passing_spec_name]' passed
(0.271 seconds).

-[SpecSuiteName failling_spec_name]
  Test Case '-[SpecSuiteName failling_spec_name]' started.
  Test Case '-[SpecSuiteName failling_spec_name]' failed
(0.002 seconds).

(…)

Executed 2 tests, with 1 failure (1 unexpected) in 0.273
(0.278) seconds

2 tests; 0 skipped; 1 failure; 1 exception; 0 pending
```

Run your tests from command line.

Seriously, do.
It’s pretty awesome.

“Perfect” setup:
Have your tests run each time you change something in a file.

Tip:
Use xctool



AppCode

“AppCode definitely empowers TDD. What I didn't get until I saw someone's screencast is to really lean on Extract Variable to reduce typing.”

Jon Reid

reduce typing

Resources & Contact

Code Examples
github.com/paweldudek

Contact
@eldudi
pawel@dudek.mobi