

# 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

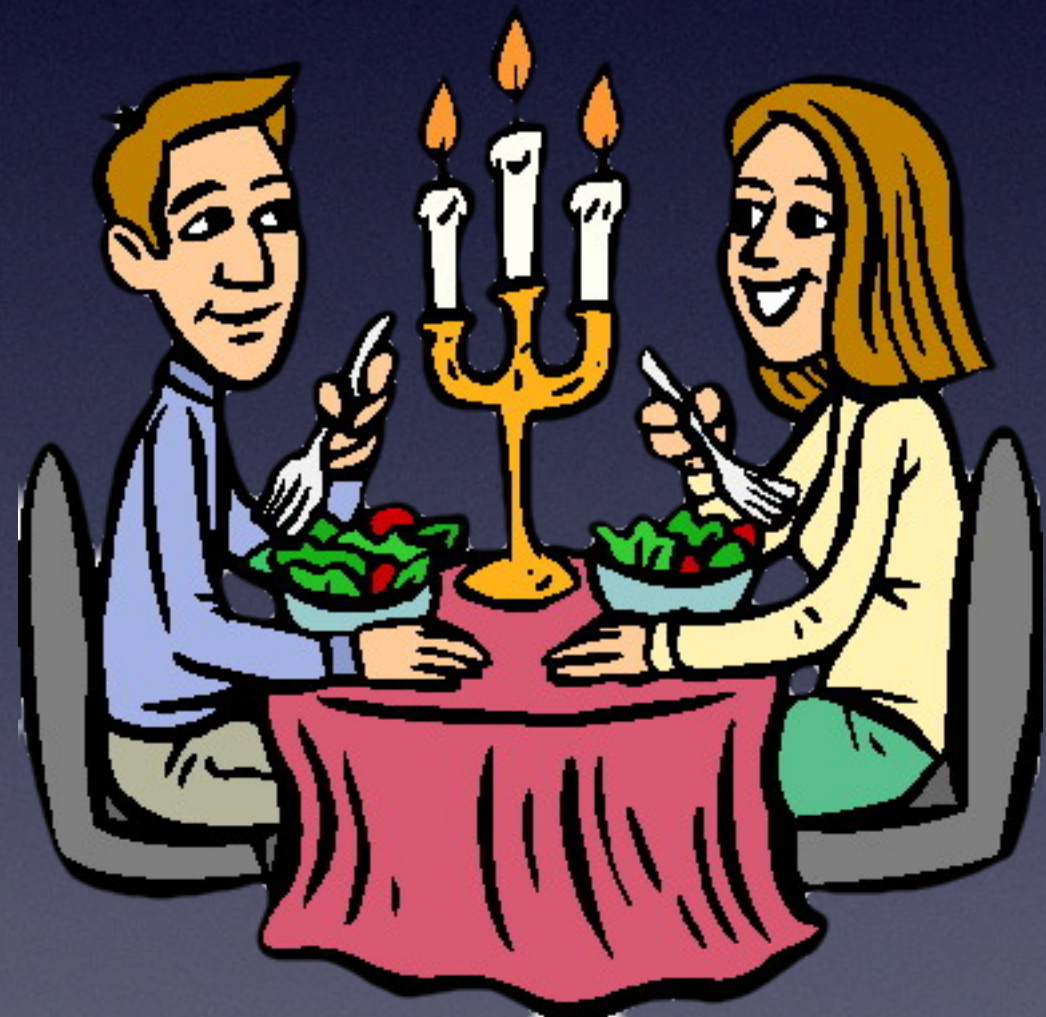


Unit tests test smallest  
parts of your code in  
**isolation** with test code



# Test isolation





Table

processOrder



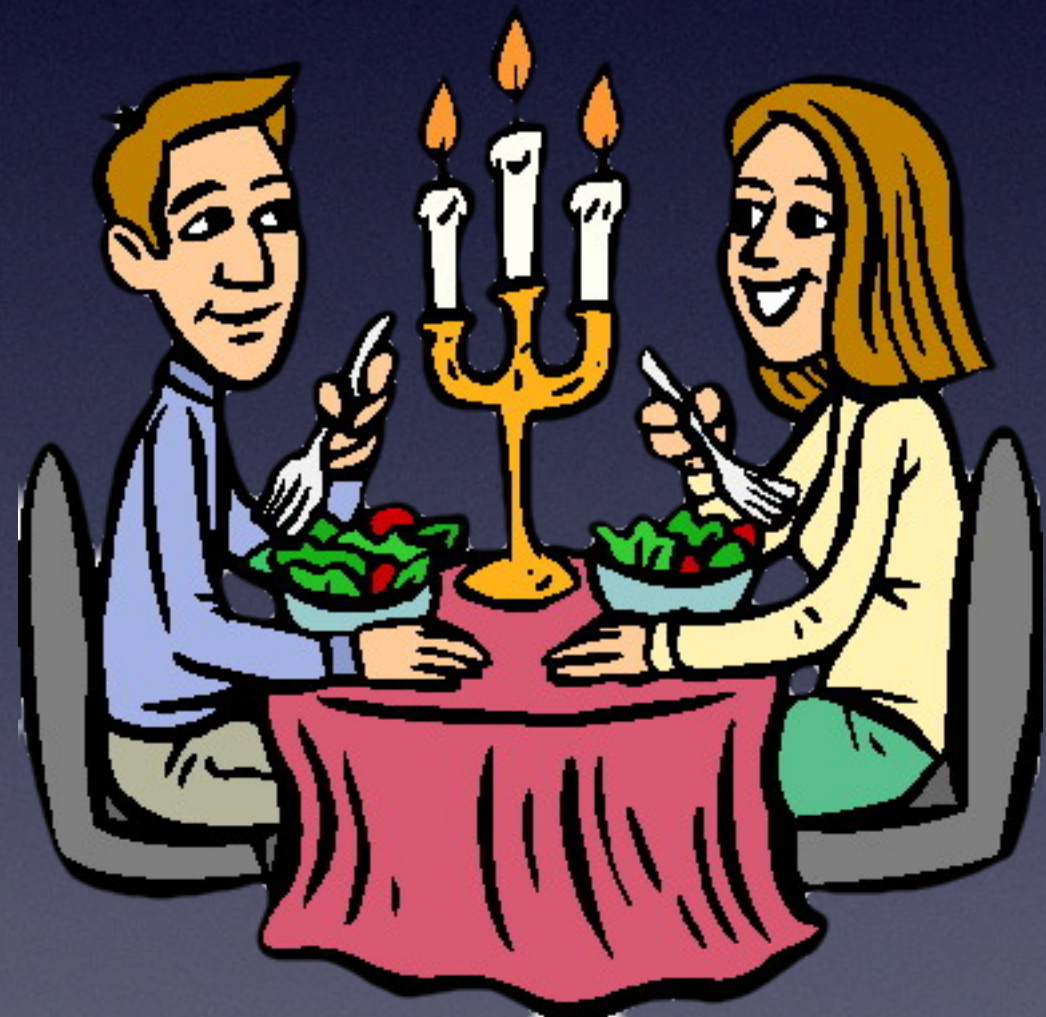
Waiter

getDishes



Cook





Table

processOrder



Waiter

getDishes



Fake cook



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.



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.



By writing test first you're becoming a consumer of your upcoming API.



Is that API hard to test?



That means it will probably  
be hard to consume too.



Or it will be really hard to  
maintain.



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



# Specta



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.

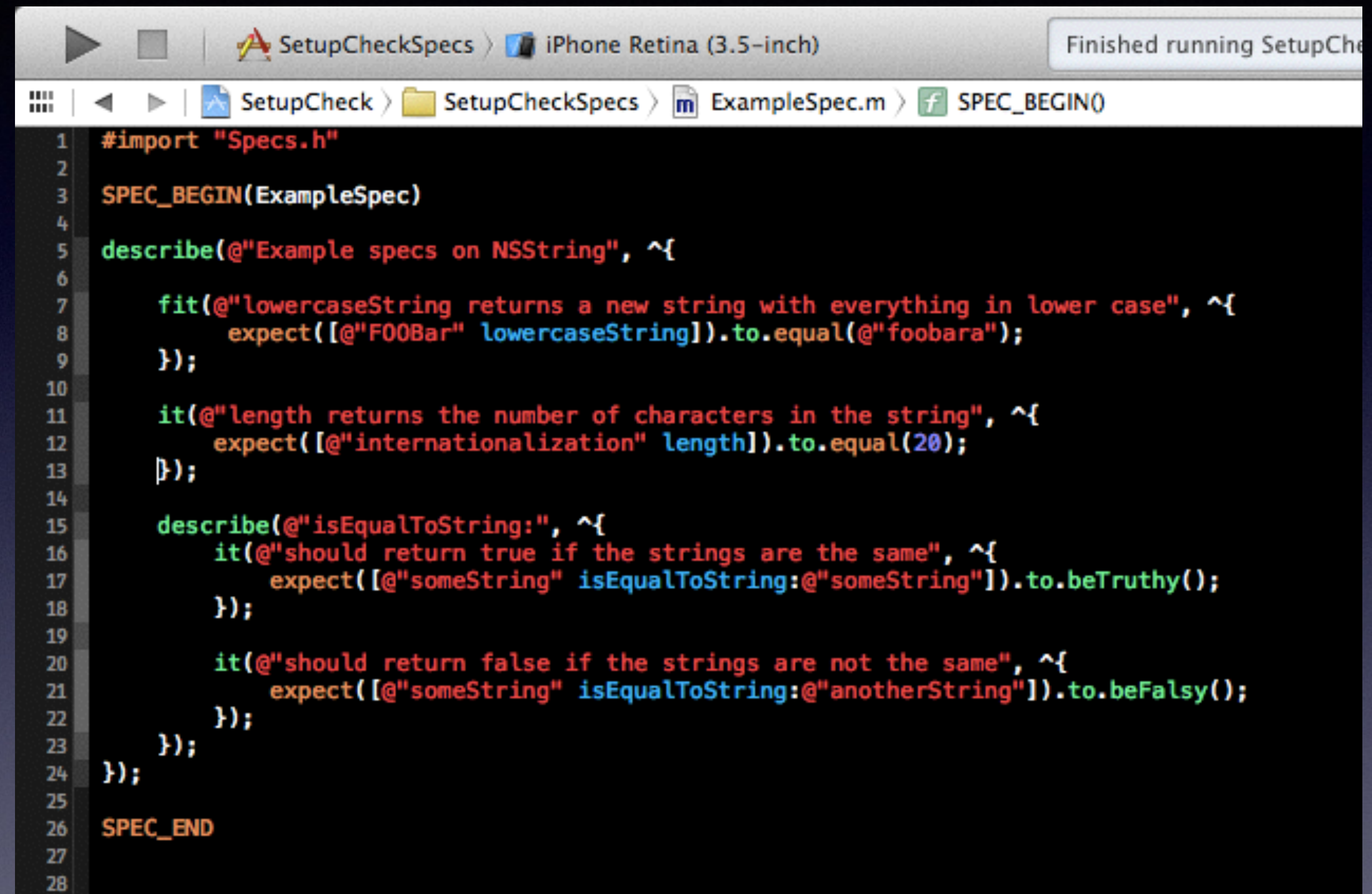
And isolate behaviour for  
different scenarios.



```
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.



The screenshot shows a code editor window with a title bar that includes a play button, a square icon, and the text "SetupCheckSpecs > iPhone Retina (3.5-inch)". A button in the top right corner says "Finished running SetupChe". The breadcrumb navigation shows "SetupCheck > SetupCheckSpecs > ExampleSpec.m > SPEC\_BEGIN()". The code is as follows:

```
1 #import "Specs.h"
2
3 SPEC_BEGIN(ExampleSpec)
4
5 describe(@"Example specs on NSString", ^{
6
7     fit(@"lowercaseString returns a new string with everything in lower case", ^{
8         expect([@"FOOBar" lowercaseString]).to.equal(@"foobara");
9     });
10
11     it(@"length returns the number of characters in the string", ^{
12         expect([@"internationalization" length]).to.equal(20);
13     });
14
15     describe(@"isEqualToString:", ^{
16         it(@"should return true if the strings are the same", ^{
17             expect([@"someString" isEqualToString:@"someString"]).to.beTruthy();
18         });
19
20         it(@"should return false if the strings are not the same", ^{
21             expect([@"someString" isEqualToString:@"anotherString"]).to.beFalsy();
22         });
23     });
24 });
25
26 SPEC_END
27
28
```



Before/After each  
blocks



```
beforeEach(^{  
    appDelegate = [[AppDelegate alloc] init];  
});  
  
afterEach(^{  
    appDelegate = nil;  
});  
  
it(@"should have a window", ^{  
    expect(appDelegate.window).to.beKindOfClass([UIWindow class]);  
});
```



1

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

2

```
it(@"should have a window", ^{  
    expect(appDelegate.window).to.beKindOfClass([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.



Enhance your tests output.



(...)

–[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







xctool vs xcpretty



# 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/mobile-warsaw](https://github.com/mobile-warsaw)

## Contact

[@eldudi](#)

[pawel@dudek.mobi](mailto:pawel@dudek.mobi)