Unit Testing that Doesn't Suck

Daniel H Steinberg
http://dimsumthinking.com

Our Path

- Quickly create a project and add Kiwi to it
- Use concrete examples of Behavior Driven Development to drive development of a simple RPN calculator

To begin quickly, create a new project named CocoaCalcTests with Unit Tests and drag the Kiwi project in.

We begin

In our first Kiwi test a sample string should be initialized.

- We will usually throw out the generated test files
- For now we'll ignore the .h file and replace the contents of the CocoaCalcTests.m file
- This pass through is just to explore syntax
- The next pass will explore meaning

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
'In our first Kiwi test a
sample string should be
initialized' [PASSED]
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"

SPEC_BEGIN(CocoaCalcTests)
describe(@"In our first Kiwi test", ^{
    context(@"a sample string", ^{
        NSString *greeting = @"Hello, World!";
        it(@"should be initialized", ^{
            [greeting shouldBeNil];
        });
    });
});

SPEC_END
```

```
'In our first Kiwi test a
sample string should be
initialized' [FAILED],
expected subject to be nil,
got "Hello, World!"
```

- What do we mean by describe, context, it,...
- Throw out our CocoaCalcTests.h and .m
- Create a new test class named StackSpec
- Throw out StackSpec.h
- Replace the contents of StackSpec.m with...

```
#import "Kiwi.h"

SPEC_BEGIN(StackSpec)

SPEC_END
```

StackSpec is the
name of the class

```
#import "Kiwi.h"

SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{

});

SPEC_END
```

Describe names the object we're interested in.

```
#import "Kiwi.h"

SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{
    context(@"at startup", ^{

    });
    context(@"when it contains only the number 3", ^{
    });
});

SPEC_END
```

The context helps us understand the state of the object we're testing

```
#import "Kiwi.h"

SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{
    context(@"at startup", ^{
        it(@"is not nil", ^{

        });
        it(@"is empty", ^{

        });

    });
});

SPEC_END
```

We use it to describe what we are actually testing about this object in this context.

```
#import "Kiwi.h"

SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{
    context(@"at startup", ^{
        pending(@"is not nil", ^{

        });
        pending(@"is empty", ^{

        });

    });
});

SPEC_END
```

We can use pending to describe unimplemented tests.

```
'The stack at startup is not nil' [PENDING]
'The stack at startup is empty' [PENDING]
```

- Create a class named RPNStack and let's use Kiwi and BDD to flesh it out

```
#import "Kiwi.h"
#import "RPNStack.h"


SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{
    RPNStack *stack = [[RPNStack alloc] init];
    context(@"at startup", ^{
        it(@"is not nil", ^{
            [stack shouldNotBeNil];
        });
    });
});

SPEC_END
```

```
#import "Kiwi.h"
#import "RPNStack.h"


SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{
    RPNStack *stack = [[RPNStack alloc] init];
    context(@"at startup", ^{
        it(@"is not nil", ^{
            [stack shouldNotBeNil];
        });
        it(@"will allow me to add a double to it", ^{
            [stack push:5.3];
            [[stack top] shouldEqual:5.3];
        });
    });
});

SPEC_END
```

```
[[stack top] shouldEqual:5.3];
```

```
[theValue([stack top]) shouldEqual:theValue(5.3)];
```



```
[[theValue([stack top]) should] equal:5.3 withDelta:.01];
```

- At this point we'd implement top and push in RPNStack enough to make our tests pass

```
#import <Foundation/Foundation.h>

@interface RPNStack : NSObject

- (void) push:(double) numberToPush;
- (double) top;
@end
```

```objc
#import "RPNStack.h"

@interface RPNStack()
@property(strong) NSMutableArray *stack;
@end

@implementation RPNStack
@synthesize stack = _stack;

- (id) init {
    if (self = [super init]) {
        _stack = [[NSMutableArray alloc] initWithCapacity:4];
    }
    return self;
}
- (void) push:(double) numberToPush {
    [self.stack addObject:[NSNumber numberWithDouble:numberToPush]];
}
- (double) top {
    return [[self.stack lastObject] doubleValue];
}
@end
```

- I'd love to test that the stack
- is empty at startup (I've tested it's not nil) and
- contains one item after I push a double onto it

- Depending on your deeply held tdd/bdd religious beliefs
  - you will create a public method named count and use that for testing the number of elements on the stack - or -
  - you will take advantage of an Obj-C mechanism for inspecting the internals

```objc
#import "RPNStack.h"

@interface RPNStack()
@property(strong) NSMutableArray *stack;
@end

@implementation RPNStack
@synthesize stack = _stack;

- (id) init {
    if (self = [super init]) {
        _stack = [[NSMutableArray alloc] initWithCapacity:4];
    }
    return self;
}
- (void) push:(double) numberToPush {
    [self.stack addObject:[NSNumber numberWithDouble:numberToPush]];
}
- (double) top {
    return [[self.stack lastObject] doubleValue];
}
@end
```

```
#import "Kiwi.h"
#import "RPNStack.h"

@interface RPNStack(TestingMethodsForStackSpec)
@property(strong, readonly) NSMutableArray *stack;
@end

SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{
    RPNStack *stack = [[RPNStack alloc] init];
    context(@"at startup", ^{
        it(@"is not nil", ^{
            [stack shouldNotBeNil];
        });
        it(@"will be empty", ^{
            [[stack.stack should] beEmpty];
        });
        it(@"will allow me to add a double to it", ^{
            [stack push:5.3];
            [[theValue([stack top]) should] equal:5.3 withDelta:.01];
        });
    });
});
SPEC_END
```

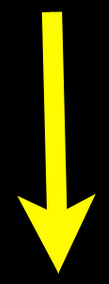
- Wait a minute - it looks as if order of our tests matter...

```
#import "Kiwi.h"
#import "RPNStack.h"

@interface RPNStack(TestingMethodsForStackSpec)
@property(strong, readonly) NSMutableArray *stack;
@end

SPEC_BEGIN(StackSpec)

describe(@"The stack", ^{
    RPNStack *stack = [[RPNStack alloc] init];
    context(@"at startup", ^{
        it(@"is not nil", ^{
            [stack shouldNotBeNil];
        });
        it(@"will allow me to add a double to it", ^{
            [stack push:5.3];
            [[theValue([stack top]) should] equal:5.3 withDelta:.01];
        });
        it(@"will be empty", ^{
            [[stack.stack should] beEmpty];
        });
    });
});
SPEC_END
```

- Hey, xunit lets us do setUp and tearDown what about Kiwi?
- Actually, Kiwi allows more granular setUp and tearDown

```
SPEC_BEGIN(StackSpec)
describe(@"The stack", ^{
    RPNStack *stack = [[RPNStack alloc] init];
    afterEach(^{
        [stack clear];
    });
    context(@"at startup", ^{
        //...
    });
    context(@"after adding the item 4.7", ^{
        beforeEach(^{
            [stack push:4.7];
        });
        it(@"has one element", ^{
            [[stack.stack should] haveCountOf:1];
        });
        it(@"has 4.7 as the top", ^{
            [[theValue([stack top]) should] equal:4.7 withDelta:.01];
        });
        it(@"should be able to pop 4.7 and be empty", ^{
            [[theValue([stack pop]) should] equal:4.7 withDelta:.01];
            [[stack.stack should] beEmpty];
        });
        it(@"should return 0 when I pop an empty stack", ^{
            [stack pop];
            [[theValue([stack pop]) should] equal:0 withDelta:.01];
            [[stack.stack should] beEmpty];
        });

    });
});
SPEC_END
```

```
it(@"should return 0 when I pop an empty stack", ^{
    [stack pop];
    [[theValue([stack pop]) should] equal:0 withDelta:.01];
    [[stack.stack should] beEmpty];
});
```



```
it(@"throws an exception when I pop an empty stack", ^{
    [[theBlock(^{
                    [stack pop];
                    [stack pop];
                }) should] raise];
});
```

- Let's assume we get the stack working and we get our Calculator Model working with methods like add, subtract, multiply, divide, squareRoot, square,...
- What about the GUI?

- Create RPNCalculatorViewController
- We'll start by testing number entry
- We need to simulate some number buttons, a decimal point button, and a display

```objc
#import <Cocoa/Cocoa.h>

@interface RPNCalculatorViewController : NSViewController
@end
```

```
#import "RPNCalculatorViewController.h"

@interface RPNCalculatorViewController()
@property(assign) NSTextField *display;
@end


@implementation RPNCalculatorViewController
@synthesize display = _display;

- (IBAction)userDidPressNumber:(NSButton *)numberButton {
}
- (IBAction)userDidPressDecimalPoint:(NSButton *)decimalPointButton {
}
@end
```

```objc
#import "Kiwi.h"
#import "RPNCalculatorViewController.h"

@interface RPNCalculatorViewController(TestingMethodsForNumberEntrySpec)
@property(assign) NSTextField *display;
- (IBAction)userDidPressNumber:(NSButton *)numberButton;
- (IBAction)userDidPressDecimalPoint:(NSButton *)decimalPointButton;
@end

SPEC_BEGIN(NumberEntrySpec)
```

```
describe(@"Calculator View Controller", ^{
    context(@"when the display is blank", ^{
        RPNCalculatorViewController *calcVC
                    = [[RPNCalculatorViewController alloc]init];
        NSButton *numberThreeButton = [[NSButton alloc] init];
        numberThreeButton.title = @"3";
        NSButton *numberFiveButton = [[NSButton alloc] init];
        numberFiveButton.title = @"5";
        NSButton *decimalPointButton = [[NSButton alloc] init];
        decimalPointButton.title = @".";
        calcVC.display = [[NSTextField alloc] init];
        beforeEach(^{
            calcVC.display.stringValue = @"";
            [decimalPointButton setEnabled:YES];
        });
```

```
it(@"should display 3 when the 3 button is pressed", ^{
            [calcVC userDidPressNumber:numberThreeButton];
            [[calcVC.display shouldNot] beNil];
            [[calcVC.display.stringValue should] equal:@"3"];
        });
it(@"displays 35 when the 3 and 5 buttons are pressed", ^{
    [calcVC userDidPressNumber:numberThreeButton];
    [calcVC userDidPressNumber:numberFiveButton];
    [[calcVC.display.stringValue should] equal:@"35"];
});
```

```
it(@"displays 3.5 when the 3, ., and 5 numbers are pressed", ^{
    [calcVC userDidPressNumber:numberThreeButton];
    [calcVC userDidPressDecimalPoint:decimalPointButton];
    [calcVC userDidPressNumber:numberFiveButton];
    [[calcVC.display.stringValue should] equal:@"3.5"];
});
it(@"should disable the . button after it is first pressed", ^{
    [calcVC userDidPressNumber:numberThreeButton];
    [calcVC userDidPressDecimalPoint:decimalPointButton];
    [calcVC userDidPressNumber:numberFiveButton];
    [[theValue([decimalPointButton isEnabled]) should] beNo];
});
```

```
it(@"displays 3.53 when 3, ., 5, ., and 3 are pressed", ^{
    decimalPointButton.target = calcVC;
    decimalPointButton.action = @selector(userDidPressDecimalPoint:);
    [calcVC userDidPressNumber:numberThreeButton];
    [decimalPointButton performClick:decimalPointButton];
    [calcVC userDidPressNumber:numberFiveButton];
    [decimalPointButton performClick:decimalPointButton];
    [calcVC userDidPressNumber:numberThreeButton];
    [[calcVC.display.stringValue should] equal:@"3.53"];
});
```

- Final thoughts...
  - snippets are your friends
  - think of the messages you want to see
  - alternatives to testing the GUI
  - Given, when, then syntax
  - Kiwi for Cucumber
  - Code at github.com/editorscut

Unit Testing that Doesn't Suck

Daniel H Steinberg
http://dimsumthinking.com