

Using Test Driven Development in iOS

Joe Bologna
July 27th, 2013

WHAT IS TDD?

- Simply put, it is a structured way to build your application on a solid foundation.
- The objective is to build assets that are used and re-used to build bigger and better things.
- By building a vertical slice of functionality, the structure of your application is used to incrementally improve it.
- As things progress, a version of your application will be complete for deployment.

A BASIC GUI AND DATA MODEL

- To build a vertical slide of your application, you will need a GUI. In iOS this is a UIView and UIViewController. It is usually a browser, single page, multi-page or table view. Split View is also common, but often used to provide a sliding menu vs. a tab bar for navigation.
- For simplicity, let's just consider a UIViewController with a single page.
- The example is a calculator for comparing the prices of two items, thoughtfully named "It's a Good Deal!".

THE CONCEPT & GUI

- The concept is simple, just collect 3 pieces of information about 2 items.
- Make it as easy as possible to collect the information and display a simple straightforward action to take.
- And, of course, some form of revenue. In this case, iAd with an In-App purchase to remove them.

Carrier 8:32 AM

Deal A	Deal B
Price	Price
# of Items	# of Items
# of Units Each	# of Units Each



Enter Prices, # of Items and # of Units

1	2	3	C
4	5	6	Del
7	8	9	No Ads
.	0	Next/Calc Savings	

Advertising like you've never seen or touched before. iAd

THE CALL TO ACTION

- With all of the information entered, a message is displayed showing which item to buy and how much savings the user will realize, if any.

Carrier  8:43 AM 

Deal A	Deal B
\$4.20	\$4.30
15	15
1	1
\$4.20/unit	\$4.30/unit


Buy A, You Save: \$1.50

123C

456Del

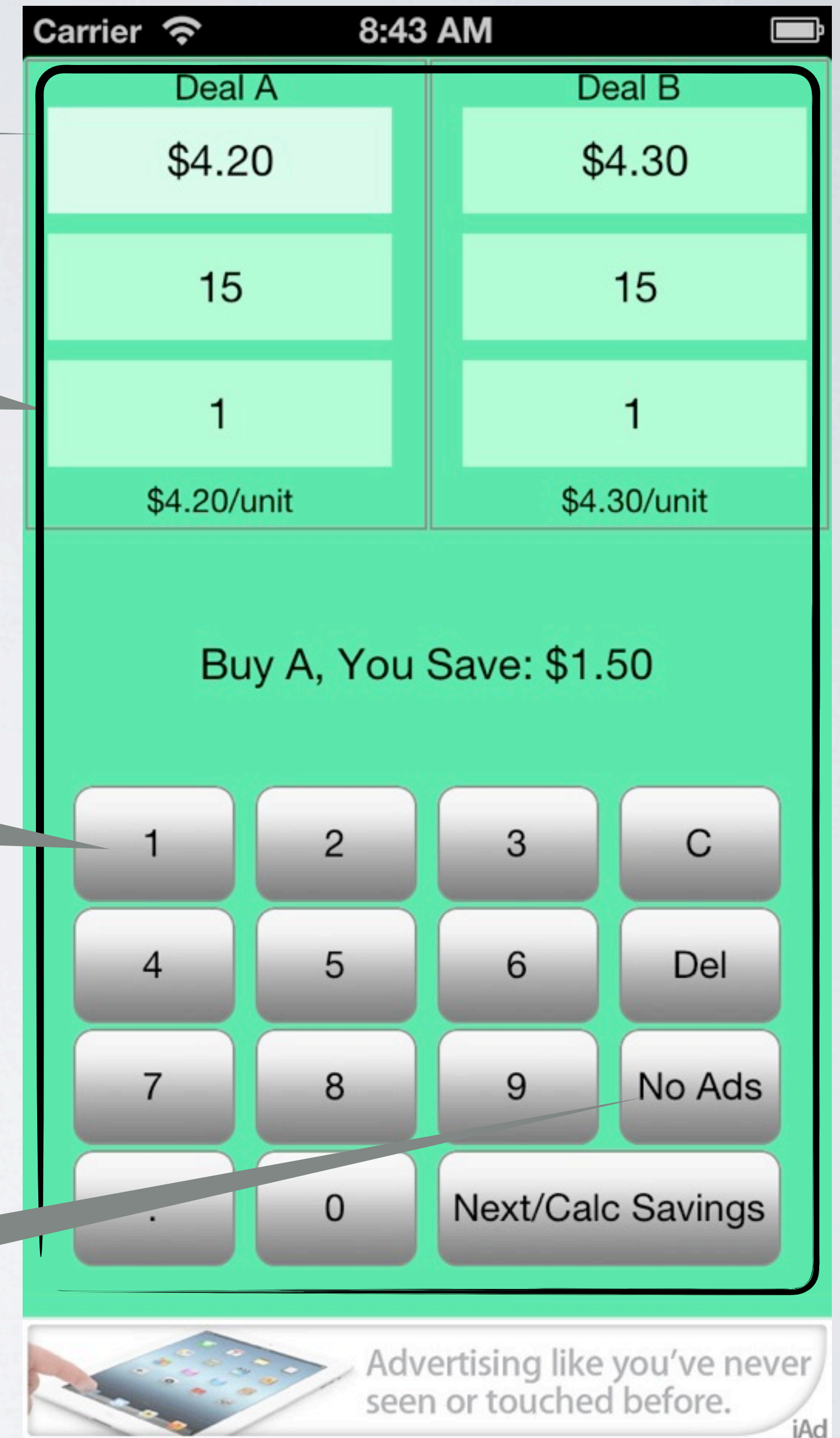
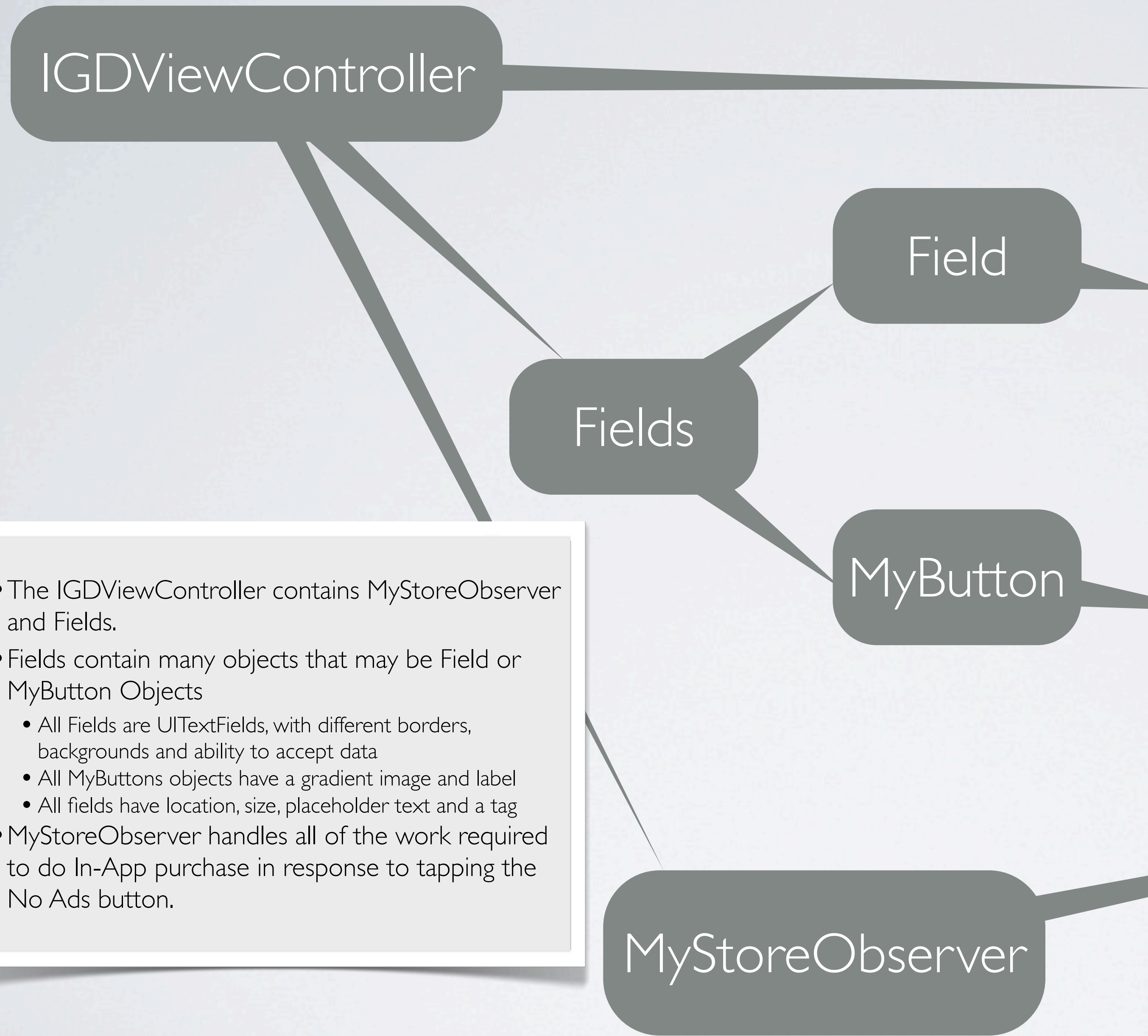
789No Ads

.0Next/Calc Savings

 Advertising like you've never seen or touched before. iAd

THE DATA MODEL

- Although it is tempting to work on making the GUI sexy first, I recommend just creating something simple and creating the data model.
 - In this case, just drop a few items on the Storyboard, give each control a name and link them up.
 - After trying a few things, I settled on the following major objects
- Field
 - Fields
 - IGDViewController
 - MyButton
 - MyStoreObserver



FIELD CENTRIC DESIGN

- I chose to make Field the most important object for handling user input and output. This is necessary because the user can tap on any field or any button at any time. The state of the field must remain consistent with the data it contains and displays. For instance, the money field must be represented internally as a number, but be displayed with a currency symbol appropriate to the region. It must also handle character deletion vs. requiring reentering the number.
- The Fields object creates all of the fields and places them on the screen, along with a handler to invoke when they are tapped.
- The Field handles user input and notifies the IGDCController or Field object to handle various situations, such as going to the next field, performing a calculation, etc.
- Field does not know about Fields or IGDCController; it just calls a method in the object to give it control. This allows Field to be a reusable object across project. The MyButton object is also an example of this.

CALCULATIONS

- Given that Field contains all of the values necessary to make the saving calculation, the savings algorithm must be implemented to use these values and provide a result.
- This is not (currently) an object, but a method in the Fields object.

OTHER NUANCES (FOR LATER)

- The Field and Fields classes take advantage of custom Setters and Getters.
- Using Setters, the value of the field is kept as a string so it can be edited, converted to a float, formatted with or without the currency symbol and assigned to the attribute to display it on the control.
- Using this method makes it easy handle each field in the same manner and test your implementation in a step wise manner.
- For testing, all classes have a toString() method implemented, which also included objects they contain.
- All methods have this construct at the top of the method:

```
#ifdef DEBUG  
NSLog(@"'%s'", __func__);  
#endif
```

- This allows for easier debugging and gets turned off automatically when the code is built for release.

TESTING THE FIELDS CLASS

- test01FieldsClass

- Tests the Fields Class by creating all fields and putting them in collections for later use.
- It simply calls makeFields and displays the result.
- A check is made to ensure the class can detect the type of device.

- test02FieldClass

- Tests the Field Class in a similar manner.

```
- (void)test01FieldsClass {
    Fields *f = [[Fields alloc] init];
    [f makeFields:(UIViewController *)self];

    NSLog(@"\n\n%@", f.toString());
    STAssertEquals(f.deviceType, iPhone5, @"Should be whatever the
simulator is set to.");

    f = [[Fields alloc] init];
    [f makeFields:nil];
    NSLog(@"\n\n%@", f.toString());
}

- (void)test02FieldClass {
    Field *f = [[Field alloc] init];
    NSLog(@"\n\n%@", f.toString());
    STAssertEquals(f.rect.origin.x, 0.0f, @"Should be Zero");
    STAssertEquals(f.rect.origin.y, 0.0f, @"Should be Zero");
    STAssertEquals(f.rect.size.width, 0.0f, @"Should be Zero");
    STAssertEquals(f.rect.size.height, 0.0f, @"Should be Zero");
    STAssertEquals(f.f, [UIFont systemFontOfSize], @"Should be
systemFontOfSize");
    STAssertTrue(f.value.length == 0, @"Should be 0");
    f = [Field allocFieldWithRect:CGRectMake(1, 1, 1, 1) andF:20
andValue:@"TestMe" andTag:ItemA andType:LabelField andVC:nil
caller:self];
    NSLog(@"\n\n%@", f.toString());
}
```


TESTING THE FIELD CLASS

- test03FieldClass

- Tests the ability for the Field Class to format strings into numbers with currency symbols.
- The last subtest checks to make sure the number is converted properly based on the type of field, ItemA is not a currency field nor a numeric field, this is a coding mistake. The unit test will fail to indicate this error has occurred.
- Note: This revealed that the Field Class does not protect itself from this issue and will supply Infinity in calculations.

```
- (void)test03FieldClass {
    NSString *c = self.currencySymbol;
    Field *f = [Field allocFieldWithRect:CGRectMake(1, 1, 1, 1) andF:20
andValue:@"0" andTag:PriceA andType:LabelField andVC:nil caller:self];
    STAssertEquals(f.floatValue, 0.0f, @"should be 0.0");
    NSLog(@"%@", [self fmtPrice:f.floatValue]);

    f.value = @".";
    STAssertEquals(f.floatValue, 0.0f, @"should be 0.0");
    NSLog(@"%@", [self fmtPrice:f.floatValue]);

    f.value = @"1";
    STAssertEquals(f.floatValue, 1.0f, @"should be 1.0");
    NSLog(@"%@", [self fmtPrice:f.floatValue]);

    f.value = [c stringByAppendingString:@"1"];
    STAssertEquals(f.floatValue, 1.0f, @"should be 1.0");
    NSLog(@"%@", [self fmtPrice:f.floatValue]);

    f.value = [c stringByAppendingString:@".1"];
    STAssertEquals(f.floatValue, 0.1f, @"should be 0.1");
    NSLog(@"%@", [self fmtPrice:f.floatValue]);

    f.value = [c stringByAppendingString:@"."];
    STAssertEquals(f.floatValue, 0.0f, @"should be 0.0");
    NSLog(@"%@", [self fmtPrice:f.floatValue]);

    f = [Field allocFieldWithRect:CGRectMake(1, 1, 1, 1) andF:20
andValue:@"0" andTag:ItemA andType:LabelField andVC:nil caller:self];
    STAssertEquals(f.floatValue, 0.0f, @"should be 0.0");
}
```


TESTING THE SAVINGS METHOD

- This is an Application Unit Test.
- The application is run and the calcSavings method is called.
- calcSavings updates the message field and unit cost fields.
- The unit test exercises all of the inputs that produce distinct outputs.
- The results are contained in the control, but not displayed in real time in the application.
- To actually tap the controls automatically, another method is required.

```
- (void)test04CalcSavings {
    vc.fields.priceA.value = vc.fields.priceB.value = @"4.20";
    vc.fields.numItemsA.value = vc.fields.numItemsB.value = @"1";
    vc.fields.unitsEachA.value = vc.fields.unitsEachB.value = @"1";
    NSLog(@"%@", vc.fields.fieldValues);
    [vc.fields calcSavings];
    STAssertTrue([vc.fields.message.value isEqualToString:@"A is the same price"]);

    vc.fields.priceA.value = @"9";
    NSLog(@"%@", vc.fields.fieldValues);
    [vc.fields calcSavings];
    STAssertTrue([vc.fields.message.value isEqualToString:@"Buy B, You Save: $4"]);

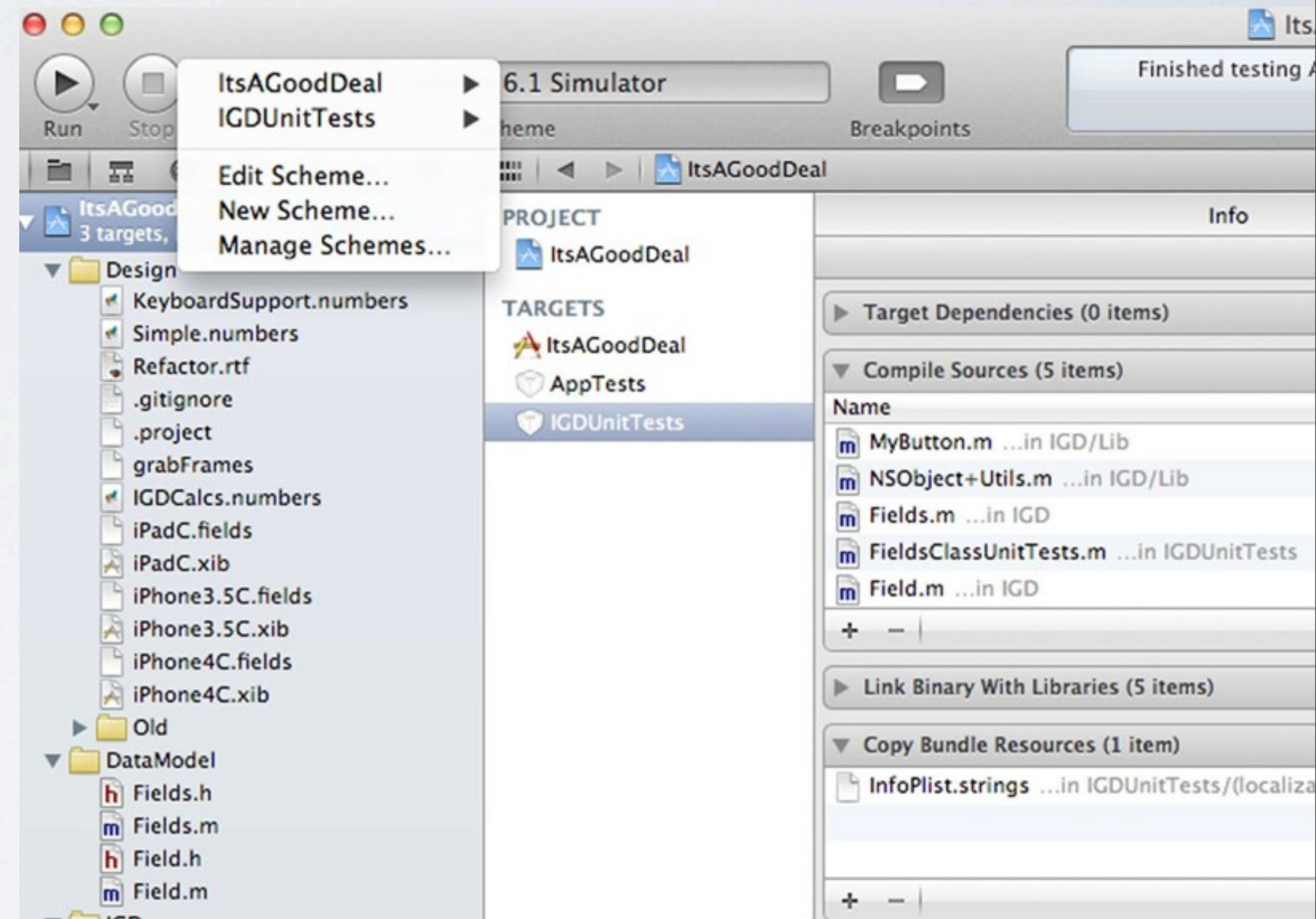
    vc.fields.priceA.value = @"1";
    vc.fields.priceB.value = @"9";
    NSLog(@"%@", vc.fields.fieldValues);
    [vc.fields calcSavings];
    STAssertTrue([vc.fields.message.value isEqualToString:@"Buy A, You Save: $8"]);

    vc.fields.priceA.value = @"1";
    vc.fields.priceB.value = @"1.011";
    NSLog(@"%@", vc.fields.fieldValues);
    [vc.fields calcSavings];
    STAssertTrue([vc.fields.message.value isEqualToString:@"Buy A, You Save: $0"]);

    ...
    STAssertTrue([vc.fields.message.value isEqualToString:@"Buy B, You Save: a"]);
}
```

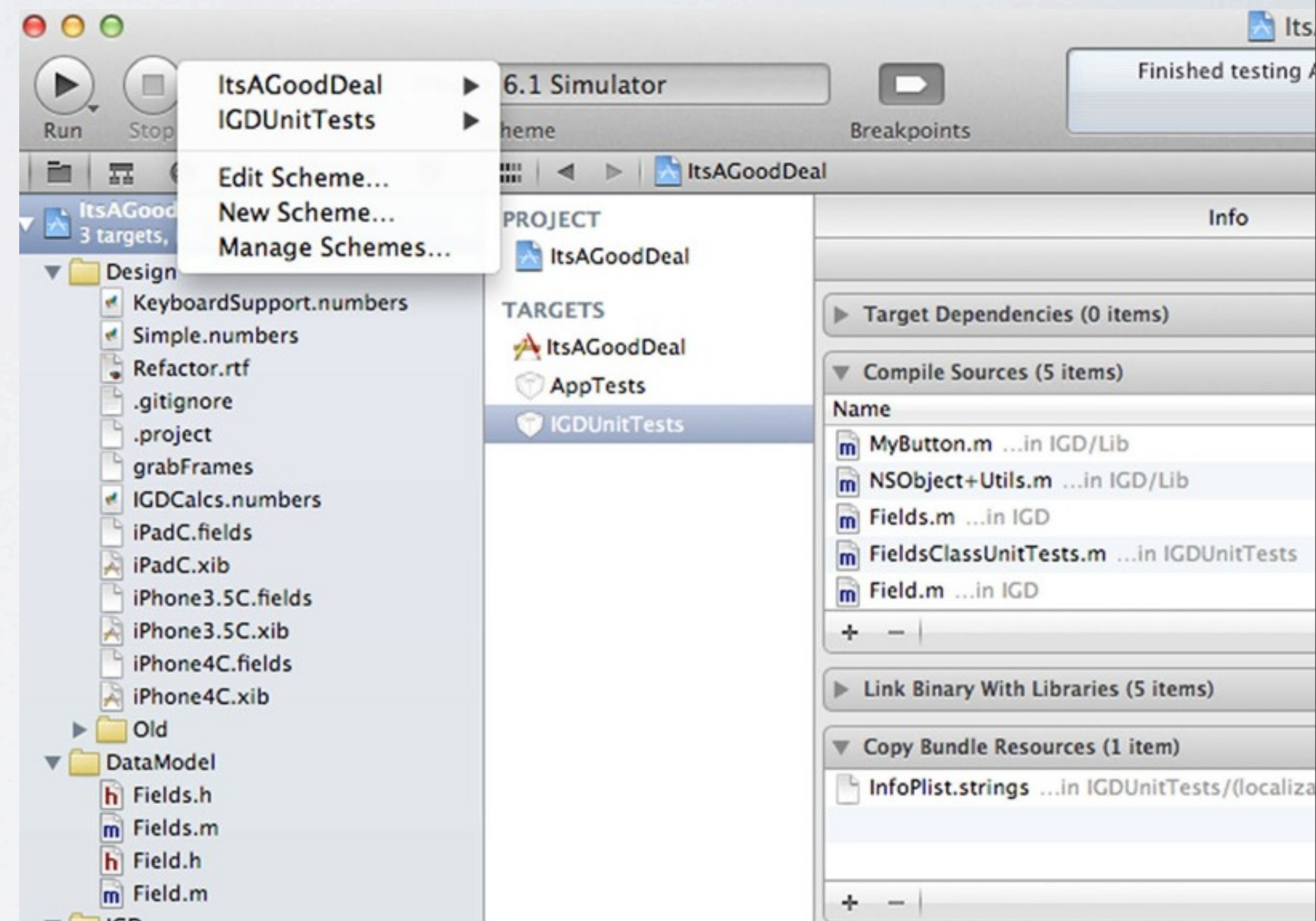

SETTING UP UNIT TESTS

- Demo of how to create a unit test target and class.
- How to run it.
- STAssert macros.



APPLICATION UNIT TESTS

- Show how to create a new target and make it dependent on the .app
- Discuss what to include in the compilation and binary
 - Don't need to recompile the app, just reference it.



UNIT TESTING REFERENCE

Setting Up Unit-Testing in a Project

This reference is online and in the Xcode documentation set.