AddOne!

Homework Project
for Xcode 4.5

Our Goals for this Project

In this project, we're going to create a very simple game called "Add One!"

In response to user actions, we'll increment a counter by one and display the results on the screen.

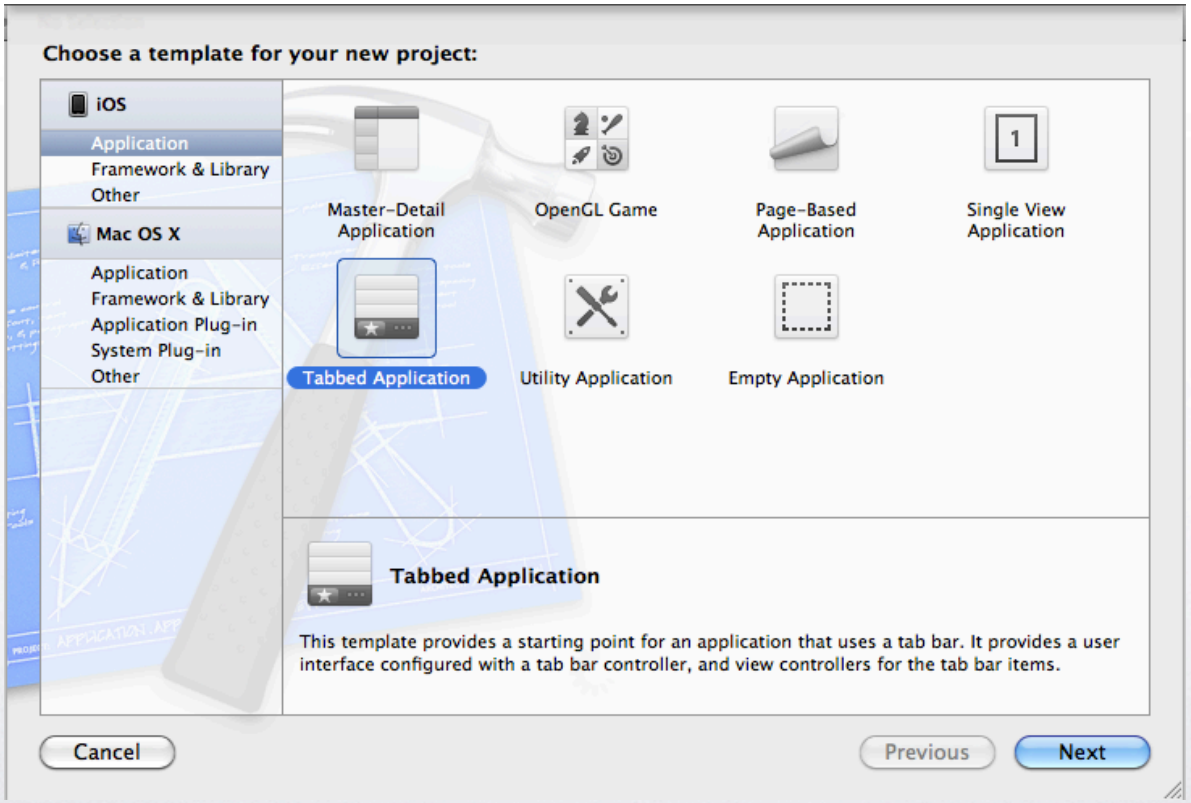
Our goal for this project is to:

- Teach you to how to create a project that uses UITabBar for navigation
- Give you experience working with an MVC app
- Give you experience working with a Singleton object
- Demonstrate a way to incorporate legacy C code into an Objective-C project
- Give you practice using Xcode to create a multi-view App
- Give you experience with -IBOutlet
- Begin to explore the different gesture recognizers available to you under iOS

© May 2011, Michael Patrick Ellard

Starting our New Project

Double-click on Xcode to open it, then select "Create New Project." Take a look at the different kinds of projects we can make. We're going to make an iOS Tabbed Application.

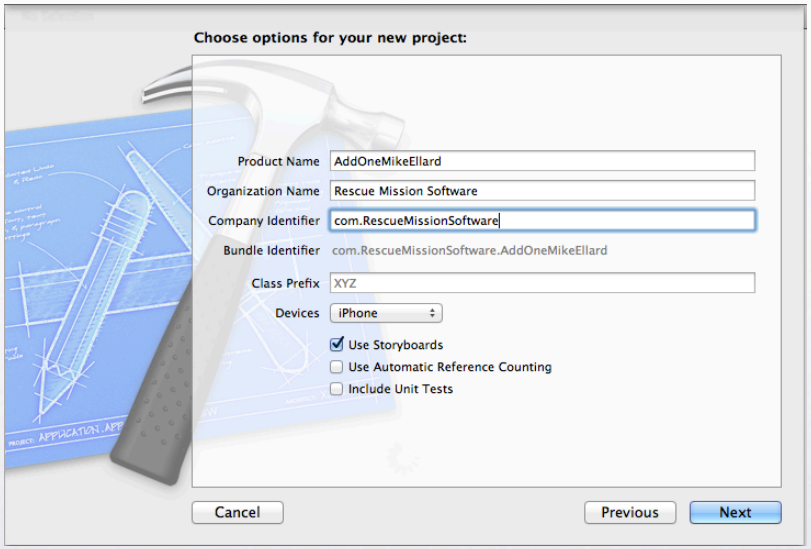


© May 2011, Michael Patrick Ellard

Naming Your Project

We're going to call this project AddOne[YourNameHere]. When naming projects for class, always include your own name in the name of the project – that will make it easier for us to tell to whom each project belongs.

Choose iPhone as the Device Family. Check the "Use Storyboard" box. Uncheck the "Use Automatic Reference Counting" and "Include Unit Tests" boxes. Leave "Class Prefix" blank.



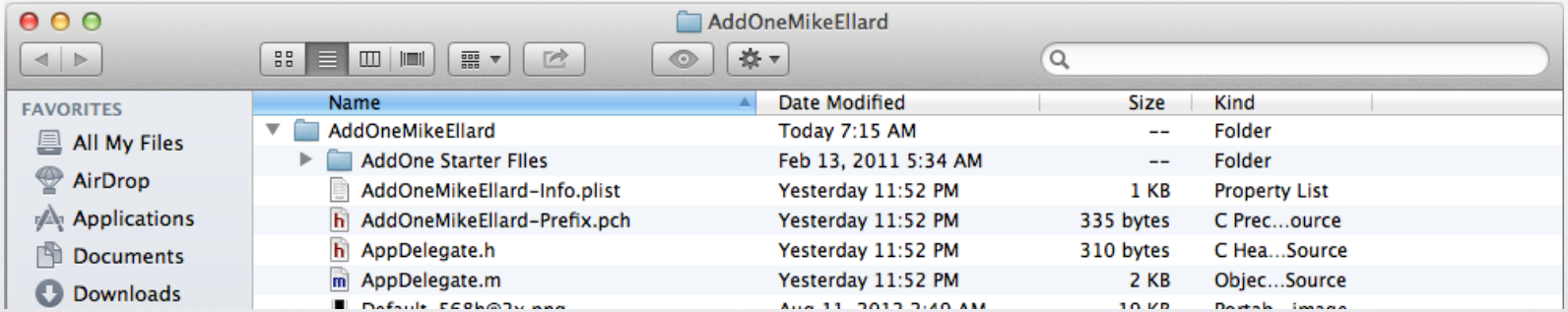
© May 2011, Michael Patrick Ellard

Moving Files to the Project Directory

Now let's go back to the Finder. Before we begin to do any work with our project, we're going to want to add some files that you'll find in the Resources section of our class website.

Copy the following AddOne Starter Files folder into your AddOne Xcode project's directory. It should contain the following files:

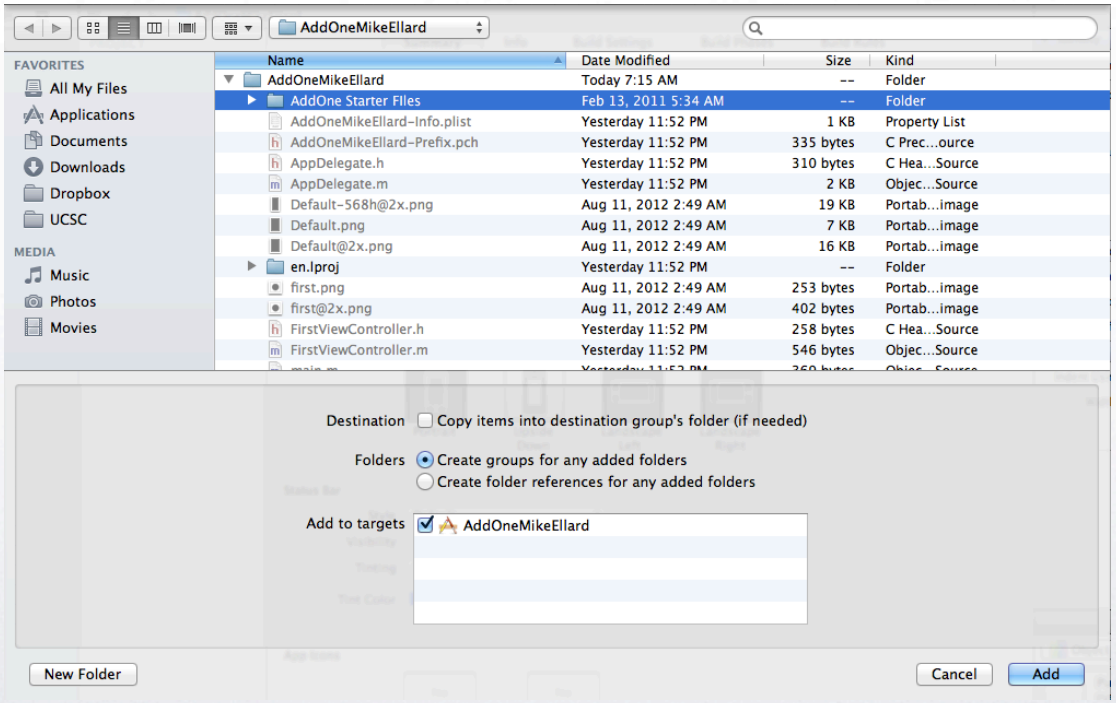
- LegacyCode.c & LegacyCode.h
- ModelObject.m & ModelObject.h
- Gradient1.png, Gradient2.png, AddOneLogo.png & KittyBackground.png
- TapIcon.png, DoubleTapIcon.png, & KittyIcon.png



© May 2011, Michael Patrick Ellard

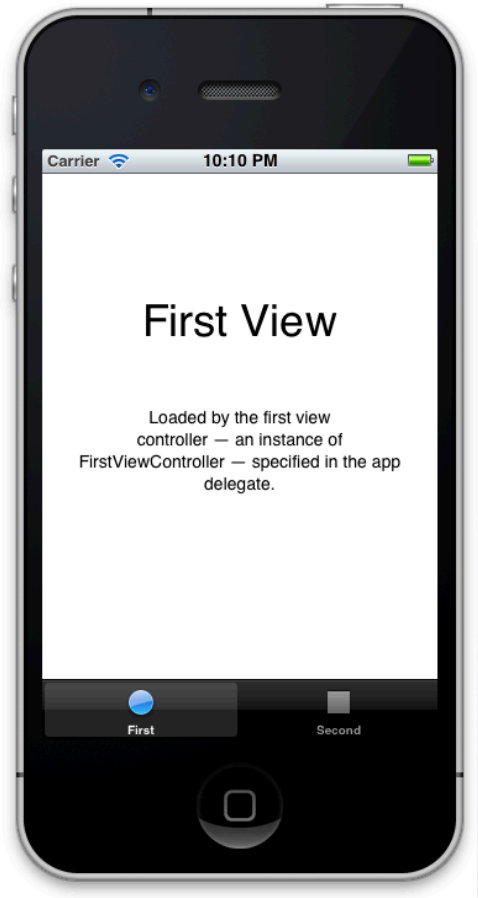
Adding Files to Our Project

Back in Xcode, select your project from the Projects Navigator at the left side of the screen and then choose the "Add Files to..." command from the File menu. Since the files are all in an enclosing folder, you can just add the folder to your project without needing to add each individual file.



© May 2011, Michael Patrick Ellard

Building Our Project



Let's run our project by click the "Run" button. Everything should build appropriately with no issues.

Our project doesn't do much at this point. We have two views with some labels on them that Xcode has generated for us, and two UITabBar buttons at the bottom of the screen that allow us to switch between our two views.

After you're done exploring, go back to Xcode and hit the Stop button.

© May 2011, Michael Patrick Ellard

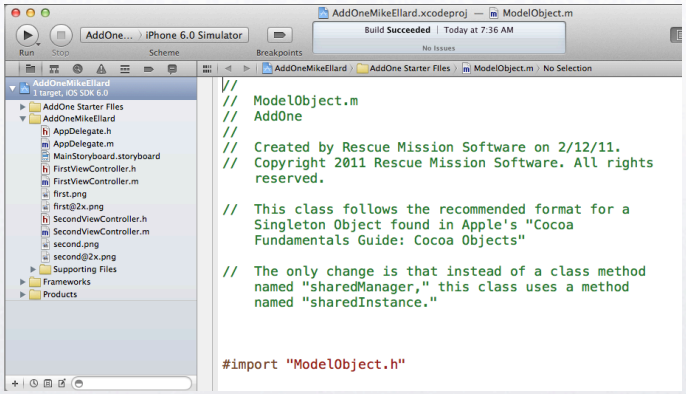
Beginning work with our Model

Click on your ModelObject.m file to view it.

This file provides the methods needed for a generic Singleton class. iOS programmers can use Singleton classes as an alternative to +alloc and -init in cases where a program will always need one, and only one, instance of a particular object.

An advantage of Singleton objects is that your other objects don't need to maintain pointers to them when they're not in use. Any object in your app can always access a Singleton object by asking the Singleton object's class for a pointer to it.

Read through the ModelObject.m file to see how it implements familiar methods such as -retain, -release, and -retainCount.



© May 2011, Michael Patrick Ellard

Updating ModelObject.h

Our model doesn't need to do much, but it does need to keep track of an integer for us and add one to it when it want it to.

Make the following updates to your ModelObject.h file:

```
@interface ModelObject : NSObject {
}
@property int currentCount;
+ (ModelObject*)sharedInstance;
- (int) addOne;
@end
```

Updates are highlighted in yellow for your convenience. They won't be yellow when you actually enter them.

© May 2011, Michael Patrick Ellard

Updating our ModelObject.m File

Now make the following updates to your ModelObject.m File

```
#import "ModelObject.h"
#import "LegacyCode.h"

@implementation ModelObject

static ModelObject *sharedModelObject = nil;

@synthesize currentCount;

#pragma mark Custom Model Methods

-(int)addOne
{
    self.currentCount = legacyAddOne(self.currentCount);

    return self.currentCount;
}

#pragma mark Housekeeping Methods

+ (ModelObject*)sharedInstance
{
    if (sharedModelObject == nil) {
        sharedModelObject = [[super allocWithZone:NULL] init];

        sharedModelObject.currentCount = 0;
    }
    return sharedModelObject;

}
```

© May 2011, Michael Patrick Ellard

Using "Legacy" C Code

You may have noticed an unfamiliar function in our updates to ObjectModel.m: legacyAddOne

Where did that come from?

That's a function from our LegacyCode.c and LegacyCode.h files. Those are "pure" C files that doesn't use any of the Objective-C extensions to C. We included them here in order to show that Objective-C and "pure" C code can easily exist together in the same project.

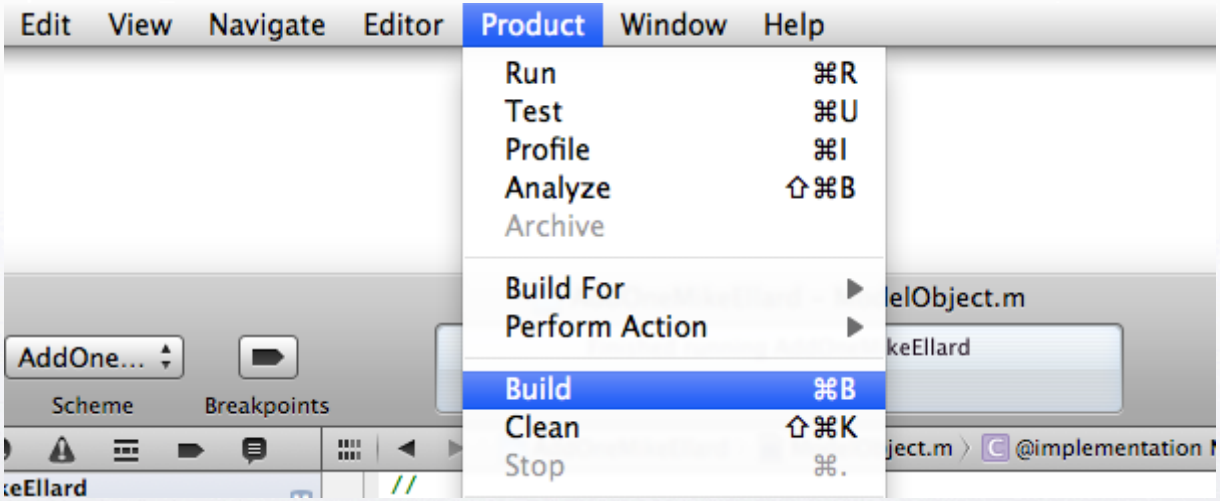
Click on LegacyCode.c and LegacyCode.h to check them out. Other than the fact that they're not Objective-C, there's not much interesting about them.

```
/*
 *  LegacyCode.c
 *  AddOne
 *
 *  Created by Rescue Mission Software on 2/12/11.
 *  Copyright 2011 Rescue Mission Software. All rights reserved.
 *
 */

#include "LegacyCode.h"


int legacyAddOne (int inputNumber)
{
    return inputNumber + 1;
}
```

© May 2011, Michael Patrick Ellard

Building Our Project

Let's build our project by choosing "Build" from Xcode's "Product" menu. Everything should build appropriately with no issues.



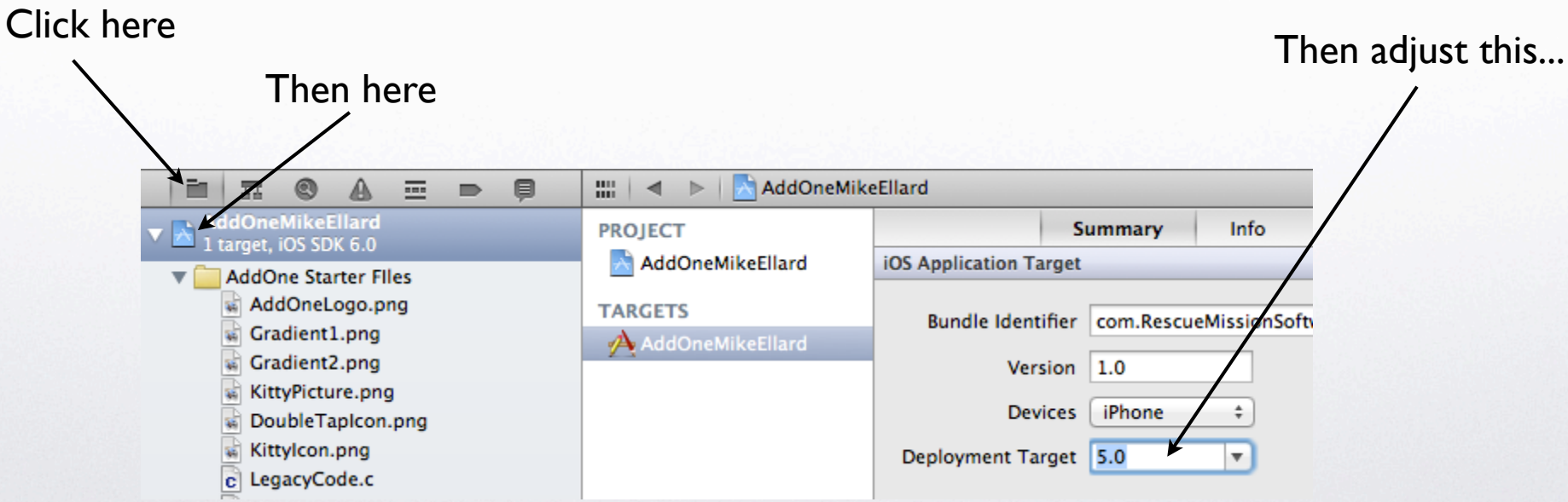
© May 2011, Michael Patrick Ellard

Changing our Deployment Target

Every app has a "Deployment Target." This is the earliest version of iOS that it will support.

Users will not be able to install our app on devices running versions of iOS earlier than the Deployment Target that we've set.

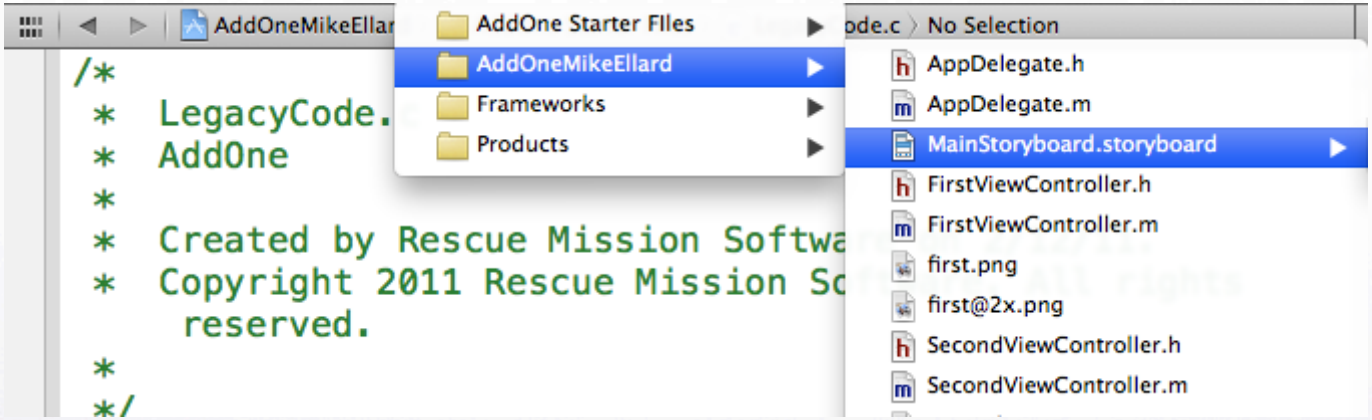
To see AddOne's Deployment Target, click on the Project navigator, then click on the project itself, then choose 5.0 from the Deployment Target menu. Our app will now run on iOS 5.



© May 2011, Michael Patrick Ellard

Finding MainStoryboard.storyboard

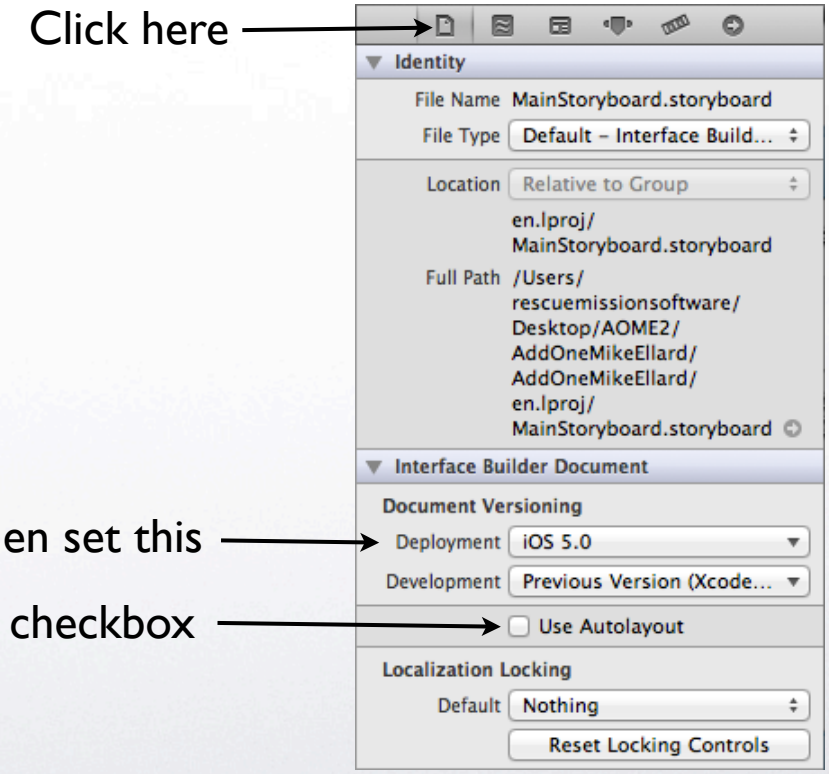
Now click on your MainStoryboard.storyboard file. If you have trouble finding it, you can use the Jump Bar to easily navigate through your project and explore its files.



© May 2011, Michael Patrick Ellard

Our Storyboard's Deployment Target

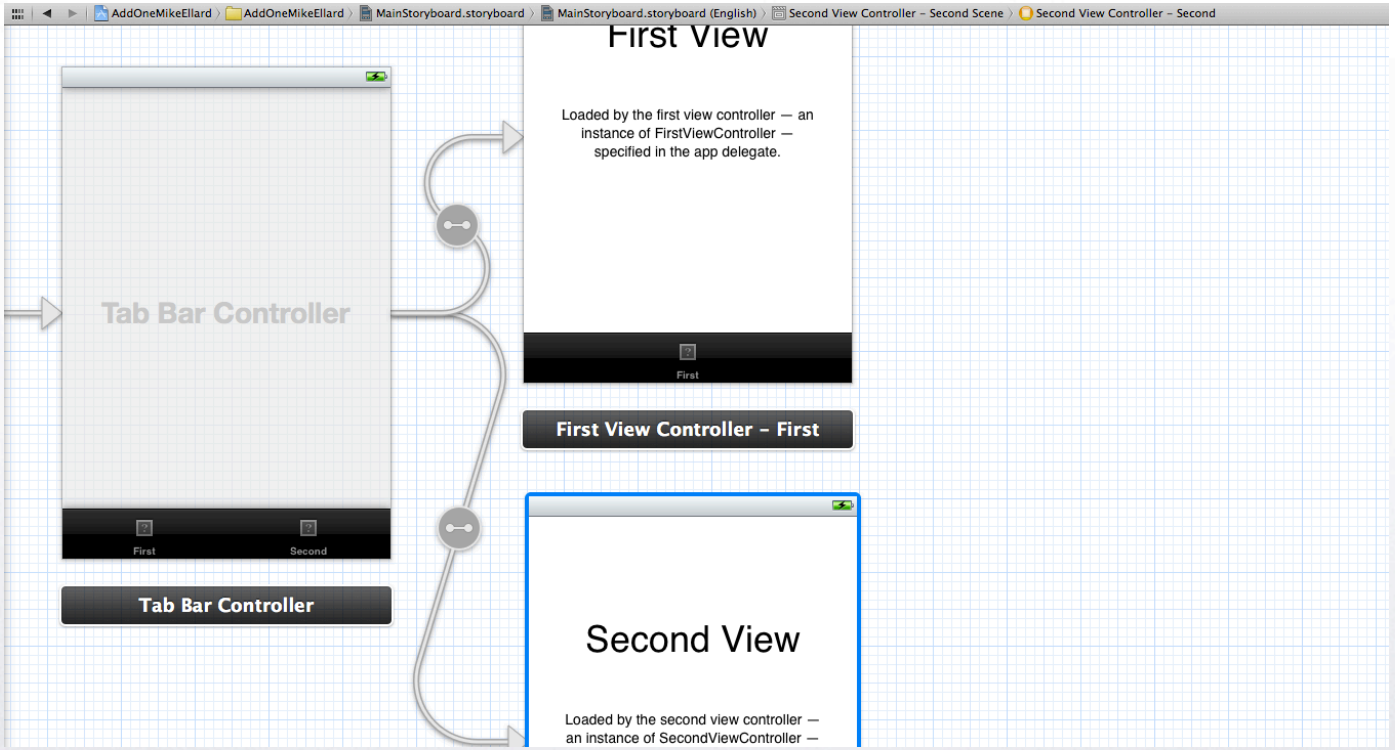
Our Storyboard also has a Deployment Target. We should change it so that it matches the Deployment Target for our app. To do this, open the Identity Inspector:



© May 2011, Michael Patrick Ellard

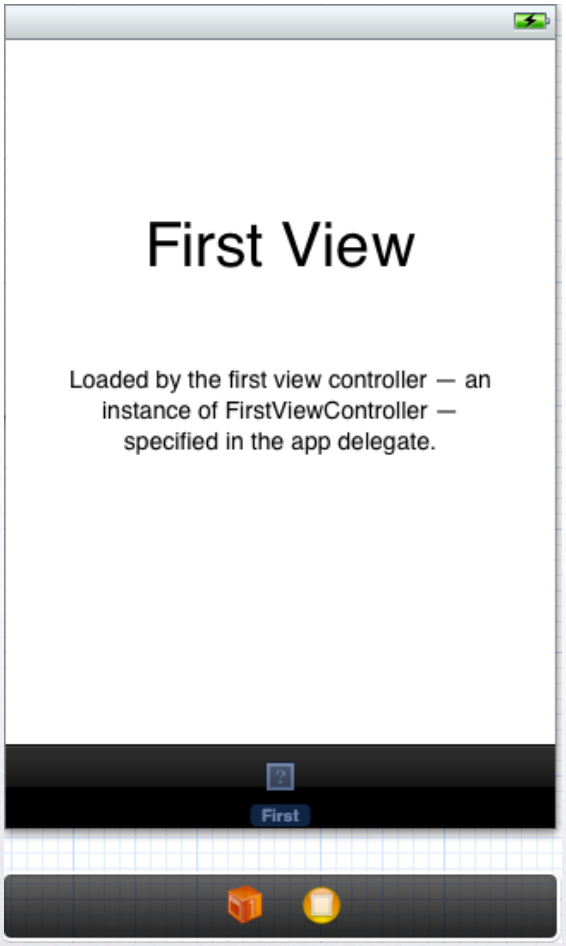
A first look at our Main Storyboard

The MainStoryboard.storyboard has three important components: A tab bar controller which will be loaded at the launch of our app, and references to two view controllers managed by the tab bar controller.



© May 2011, Michael Patrick Ellard

Setting up our Initial View

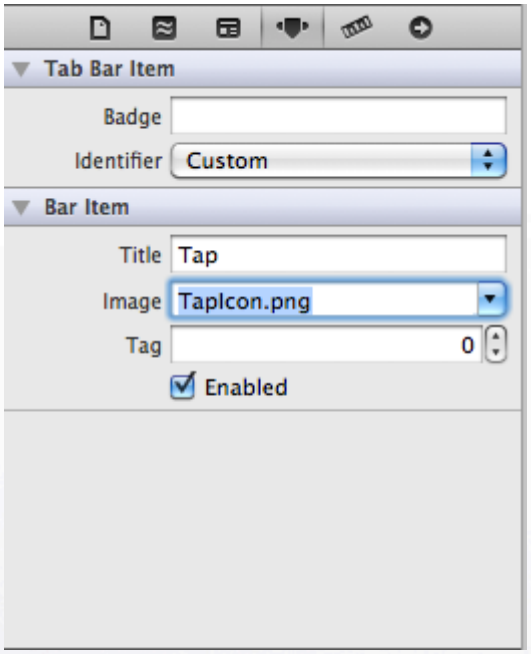


In your MainStoryboard.storyboard, click on the Tab Bar item at the bottom of the First View Controller. This is the black area with the small black square and a button saying "First."

Next, choose the "Attribute Inspector" option from the "Utilities" submenu of Xcode's "View" menu to see the options for the first Tab Bar Item.

© May 2011, Michael Patrick Ellard

Using the Attribute Inspector



First

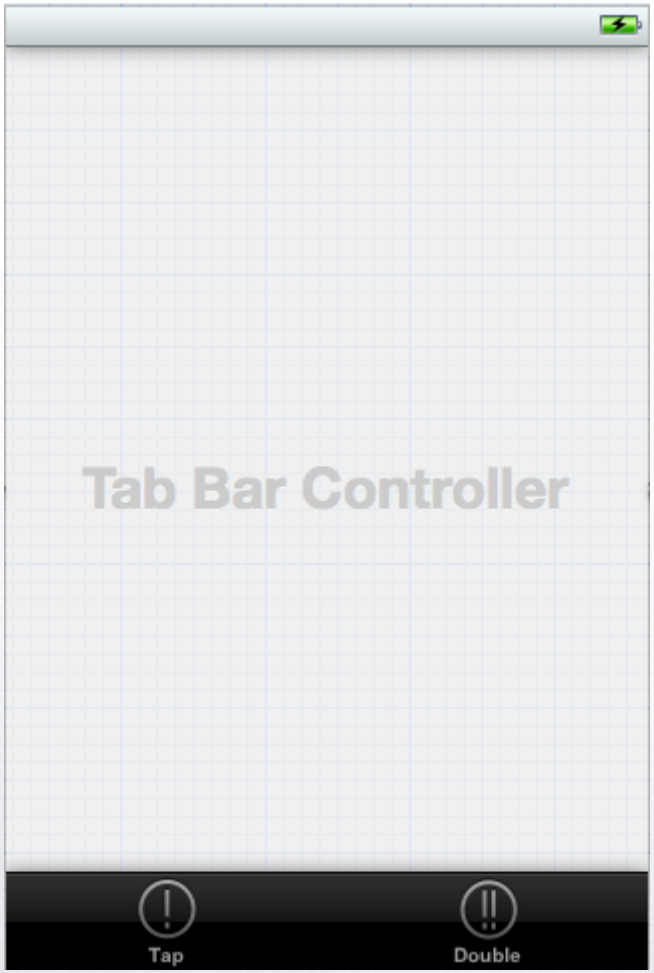
- Set the Tab Bar Item's Title to "Tap"
- Set its Image to "TapIcon.png"

Next:

- Click on the Tab Bar Item for the Second View Controller
- Set its Title to "Double"
- Set its Image to DoubleTapIcon.png.

© May 2011, Michael Patrick Ellard

Looking at the Tab Bar Controller



If you look back at the Tab Bar Controller, you should see that the changes you made to the individual view controllers' Tab Bar Items have automatically been picked up by the Tab Bar Controller: it now shows the icons and titles that you chose.

Next go to your FirstViewController.h file.

© May 2011, Michael Patrick Ellard

Updating FirstViewController.h

Make the following changes to your FirstViewController.h file.

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

@interface FirstViewController : UIViewController

@property (nonatomic, retain) IBOutlet UILabel *countLabel;

-(IBAction)addOne:(id)sender;

@end
```

© May 2011, Michael Patrick Ellard

IBOutlet & IBAction

In our edits to the header file, you may have noticed a new keyword: IBOutlet.

As you might guess, IBOutlet is closely related to another keyword that we used: IBAction

- IBAction indicates an Xcode method that we want to call from an object generated by Interface Builder. In our AirEuphonium lab, we used IBAction allow an Interface Builder-generated button to call the -makeNoise method.
- IBOutlet is essentially the same thing in reverse. It indicates an Interface Builder element that we want to be able to modify or inspect using Objective-C methods.

© May 2011, Michael Patrick Ellard

More about IBAction and IBOutlet

IBAction and IBOutlet are markers that you put in your Objective-C code to allow connections to be made between your xib or storyboard objects and your Objective-C code.

- IBAction and IBOutlet give developers the option to link code with Interface Builder objects, but the presence of these keywords does not require that developers actually create those links.
- IBAction and IBOutlet are markers used by Xcode. They don't do anything in and of themselves and your compiled Objective-C code isn't changed by including them. The only change will be in your compiled nib or storyboard files if you use IBAction or IBOutlet to connect your code with your objects.

IBAction is a preprocessor macro defined as void. IBOutlet is a preprocessor macro defined as a blank (i.e. nothing).

© November 2011, Michael Patrick Ellard

Updating FirstViewController.m

Now switch over to your FirstViewController.m file and make the following changes.

```
#import "FirstViewController.h"

// Import the header file for the Model Object class

#import "ModelObject.h"

@implementation FirstViewController

@synthesize countLabel;

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    // Get a non-retained reference to our model object

    ModelObject *myModel = [ModelObject sharedInstance];

    // Set our counter label to show the current count from our model

    self.countLabel.text = [NSString stringWithFormat:@"Count is %d", myModel.currentCount];
}

- (IBAction)addOne:(id)sender
{
    // Get a non-retained reference to our model object

    ModelObject *myModel = [ModelObject sharedInstance];

    // Call our model's addOne mehtod

    [myModel addOne];

    // Set our counter label to show the count returned from our model's currentCount method

    self.countLabel.text = [NSString stringWithFormat:@"Count is %d", myModel.currentCount];
}
```

© May 2011, Michael Patrick Ellard

viewWillAppear

One of the things we did in FirstViewController.m was to create a viewWillAppear method.

viewWillAppear is a handy UIViewController method that is called by container view controllers like UINavigationController and UITabBarController when the view for a managed view controller is about to be presented to a user. viewWillAppear is useful for doing housekeeping during view transitions, as are its similarly named siblings: viewDidAppear, viewWillDisappear, and viewDidDisappear.

We didn't need to declare viewWillAppear in our .h file, since this is an inherited method from the UIViewController class - we're just overriding it here.

The other method we created, addOne, is a custom method that we'll connect up to a button to work interactively with our view.

© May 2011, Michael Patrick Ellard

Adding a dealloc method

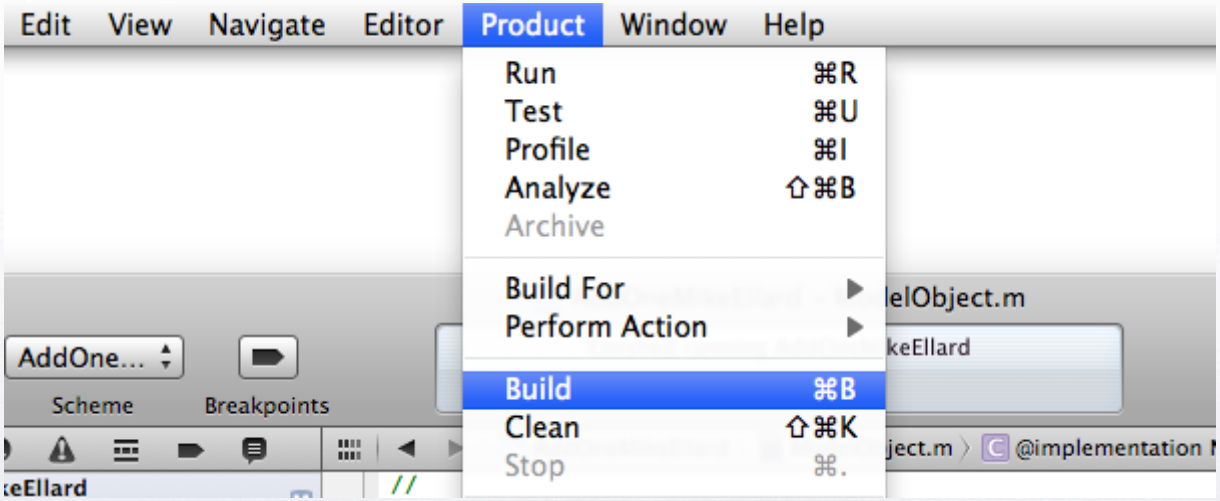
Finally, we'll need to add a dealloc method to FirstViewController.m.

```
-(void)dealloc
{
    self.countLabel = nil;
    [super dealloc];
}
```

© May 2011, Michael Patrick Ellard

Building Our Project

Let's build our project by choosing "Build" from Xcode's "Product" menu. Everything should build appropriately with no issues.



© May 2011, Michael Patrick Ellard

Creating our First View

We're going to edit our First View's layout so that it will look like the image that you see at the left.

If you're not sure where to place things, you can come back to this slide as a reference.

© May 2011, Michael Patrick Ellard

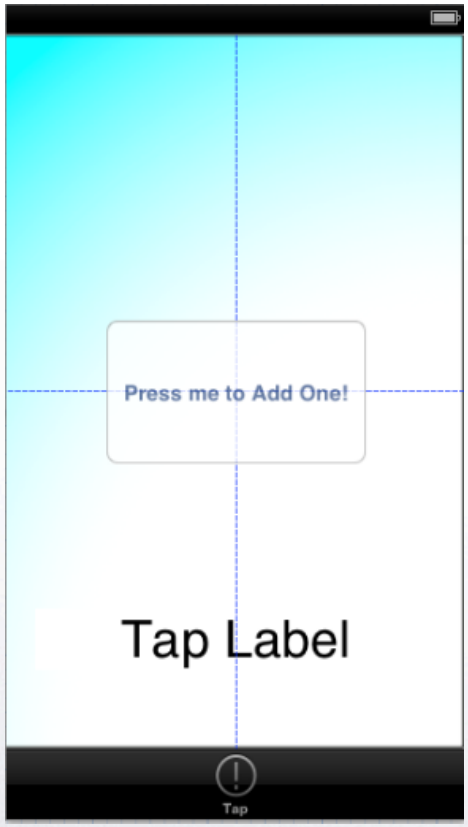
Setting up our First View

In Xcode, go the MainStoryboard and click on First View to edit it.

- Delete the big text block at the bottom of the view.
- Edit the remaining label's text to say, "Count Placeholder"
- Using the "Show Object Library" from the "Utilities" submenu of the "View" menu add a UIImageView to the screen. Make it as large as the visible screen (excluding the tab bar and the status bar).
- Use the Attributes Inspector to set the UIImageView's background to "Gradient1.png"
- We don't want our background to be in front of our label, so select the UIImageView, then choose "Send to Back" from the "Arrangement" submenu of the "Editor" menu.
- Add a button to the screen
- Edit the button's text so that it says, "Press me to Add One!"

© May 2011, Michael Patrick Ellard

Centering our Button

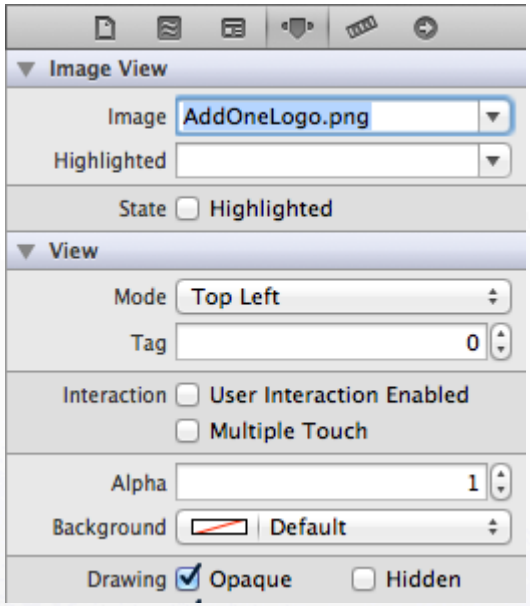


We'd like for our button to be in the center of the screen. So move it to where you think the center of the screen is. When you've got it positioned in the center, some alignment guides should appear to let you know that it's aligned in the center.

Alignment guides aren't just for centering objects, they can appear for left, right, top, bottom, vertical center, or horizontal center alignments if Xcode guesses that you're trying to line up two or more objects with one another.

© May 2011, Michael Patrick Ellard

Adding a Logo



Next, we're going to add our AddOne logo to the screen.

- Create a UIImageView and place it near the top of the screen
- Go to the Attributes Inspector and set our UIImageView's image to AddOneLogo.png

Our Logo may not look like we want - it may be stretched or compressed inappropriately. This is because of the view's Mode setting is set to "Scale to Fill" by default. This will will stretch or shrink our graphic so that it will exactly fill its enclosing UIImage View. To change the Mode setting:

- Go to the View section of the Attributes Inspector. Choose "Top Left" from the Mode menu. This will show the image at its uncompressed, unstretched size, with its top left corner at the top left corner of the UIImageView.
- If the UIImageView isn't big enough to display the whole logo, resize the UIImageView so that the whole logo is visible.

© May 2011, Michael Patrick Ellard

Constraints vs. Springs and Struts

Constraints are a powerful and flexible new feature in iOS 6. They are particularly useful for:

- Designing an app that needs to work on screens with different sizes and aspect ratios
- Designing an app with multilingual support, where the buttons or other screen elements may need to be resized to support different languages.

The biggest drawback to constraints is that they don't work on versions of iOS prior to iOS 6.

It is a good practice as a developer to support both the current version of iOS and the immediately prior version. Since we've decided to set our Deployment Target to iOS 5, we won't be able to use constraints. Instead, we'll need to use Springs and Struts, an earlier layout management system that is supported on all versions of iOS.

Springs and Struts are not as powerful and flexible as constraints, but they will still allow us to make our app automatically adjust its layout to work on both the iPhone 5 and the early models of iPhone.

© May 2011, Michael Patrick Ellard

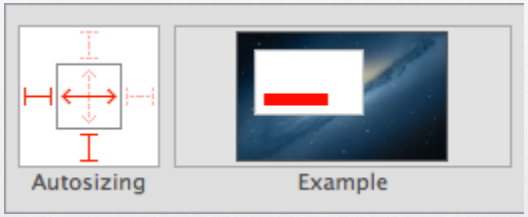
Springs and Struts

The Springs and Struts settings are shown in the Size Inspector. On the left is the Autosizing area where we specify the Springs and Struts settings. On the right is the Example area that shows how our view will change if its enclosing superview changes.

In the Autosizing controls:

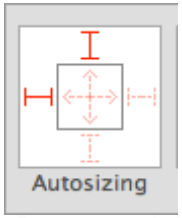
- The settings in the outer box specify the relationship of the view's frame to its enclosing view's frame.
- The inner box specifies whether the view can grow and shrink in response to changes to its enclosing view's size.

Try changing the Autosizing controls and then watch how the display changes in the Example area.

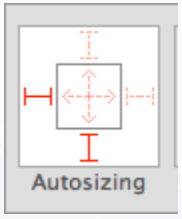


© May 2011, Michael Patrick Ellard

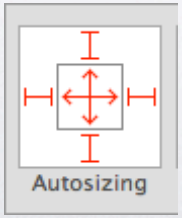
Autosizing Settings for our Screen



For the controls in our first screen, we want our logo UIImageView to always be the same distance from the top and left of our enclosing view. We don't want it to change size if its superview resizes. To do this, we should make its Autosizing properties look like the ones to the left of this paragraph.



For our Count UILabel, we want it to always be the same distance from the bottom and left of its enclosing view. We don't want it to change size if its superview resizes. To do this, we should make its Autosizing properties look like the ones to the left of this paragraph.



For our Gradient UIImageView, we want it always to be the same distance to the top, bottom, left, and right of its enclosing view. If its enclosing view resizes, we want it to grow or shrink in order to maintain the same distance to the edges of its enclosing view. To do this, we should make its Autosizing properties look like the ones to the left of this paragraph.

© May 2011, Michael Patrick Ellard

Checking our Autosizing Settings

At the bottom right of Interface Builder's main window, you will see two button groups:

This button toggles your view's size between the iPhone 5 aspect ratio and the aspect ratio used by the previous iPhones. Click on this button to make sure that your Autosizing settings are working.

This button group controls zooming. The plus button allows you to zoom in, the minus button allows you to zoom out, and the equals button allows you to toggle between actual size and your current zoom setting.

Try clicking the leftmost button to toggle your screen size. Your layout should work for both the original iPhone screen and the iPhone 5 screen.

© May 2011, Michael Patrick Ellard

First View begins to take shape!

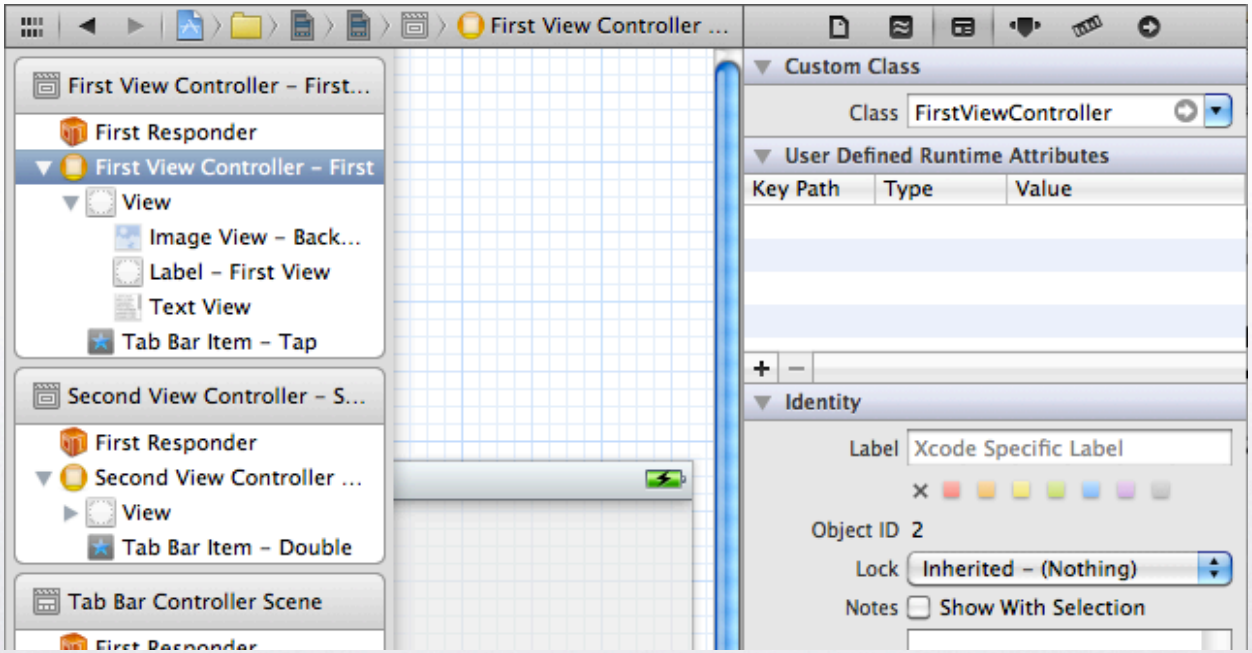
Now that we've set up our screen, it should look something like this.

Now we're ready to connect up some outlets!

© May 2011, Michael Patrick Ellard

Specifying our View's Owner

Before we can hook up our outlets, we will need to make sure that our view has the appropriate "owner." From MainStoryboard's document pane, select the owning object for the First View Controller and choose "Show Identity Inspector" from the "Utilities" submenu of the "View" menu. Make sure that the class setting is set to "FirstViewController." This lets Interface Builder know that this view's controller will be a FirstViewController object.



© May 2011, Michael Patrick Ellard

Hooking up our IBAction

Click on your button, hold down the control key, and then drag over to the First View Controller icon in the Main Storyboard document pane. A blue line should follow as you drag. Release your mouse one you've connected the blue line to the First View Controller icon, then choose the "addOne" method from the pop-up menu that appears.

© May 2011, Michael Patrick Ellard

Hooking up our IBOutlet

To hook up our label, we're going to do the same thing in reverse. Start by selecting the First View Controller icon, hold down the control key, and then drag over onto the "Count Placeholder" label. A blue line should follow as you drag. Release your mouse one you've connected the blue line to the label, then choose the "countLabel" method from the pop-up menu that appears.

© May 2011, Michael Patrick Ellard

Testing our Work

Click the "Run" button to build and run your project.

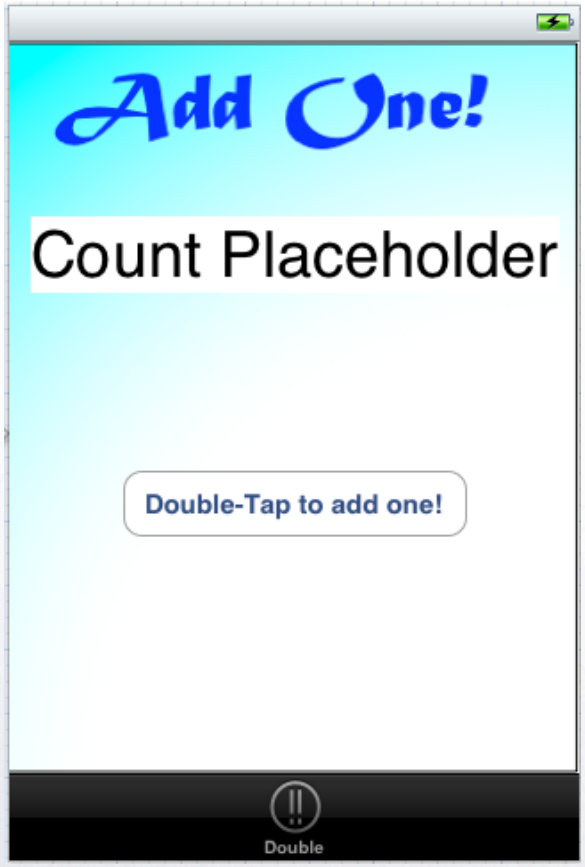
You should now see a screen like the one at the left.

When you click the button, the counter should increase.

When you're satisfied that everything is working, go back to Xcode and click the Stop button to stop execution of your app.

© May 2011, Michael Patrick Ellard

Doing it all again!



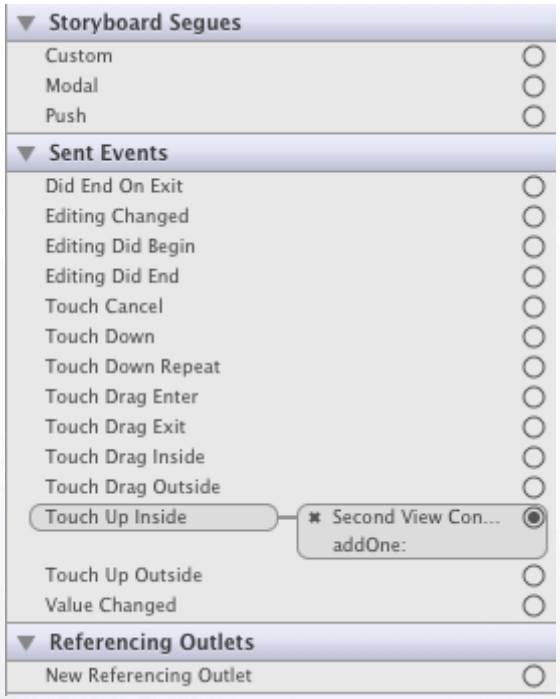
Now we want to set up our second view. The process is going to be a lot like setting up our first view. A few key differences:

- We're working with the second Tab Bar item, not the first.
- We'll use Gradient2.png, and slightly different button text. When you're done setting up your view, it should look like the one to the left.
- We're now working with a different view controller, although for now the edits are going to be exactly the same as for the first.

When you're done, build and run your project.

© May 2011, Michael Patrick Ellard

Implementing Double-taps



Everything is working great, but there's one problem. On our second screen, it says that we should double-tap to add one, and that's not really what's happening.

To solve this problem, we're going to look at a part of Xcode that we haven't explored before: the Connections Inspector.

In the SecondView in our Main Storyboard, select your button, then choose "Show Connections Inspector" from the "Utilities" submenu of Xcode's "View" menu. We can now see the kinds of events that this control can respond to.

Right now our connections are set up so that when a Touch Up Inside event occurs on our button, the addOne: method is called from the SecondViewController that owns the SecondView.

Touch Up Inside is the standard even that buttons respond to, but in this case we want Touch Down Repeat.

© May 2011, Michael Patrick Ellard

Reconnecting our Button

To change our button's connection, we need to do two things.

First we'll establish a new connection. Click in the circle to the right of Touch Down Repeat, then drag over to "Second View Controller." Select addOne: from the pop-up menu that appears. You don't need to hold down the control key, since you're starting from the Connections inspector.

After you've established the new connection, delete your old Touch Up Inside connection by clicking on the X to the right of Touch Up Inside. If you leave both connections active, addOne: will be called whenever either type of event occurs.

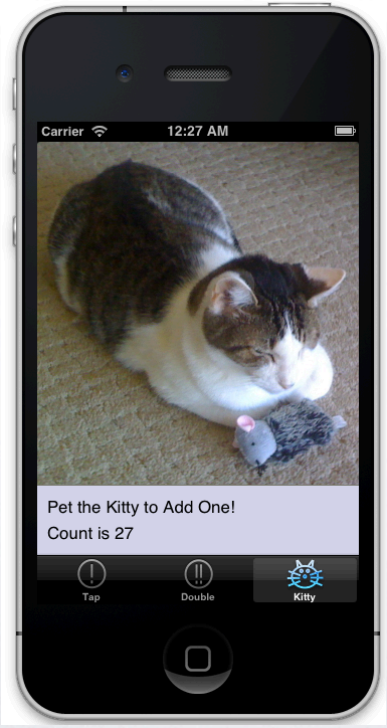
When you're done, choose "Run" to run your project and test to make sure everything is working appropriately. Double-taps should work on our second view, single-taps should work on our first view.

© May 2011, Michael Patrick Ellard

Adding a Third View

At some point, you're going to want to know how to create a Tab Bar Controller with more than just the default two views. So let's add a third view.

We're going to create a view that will look like this:

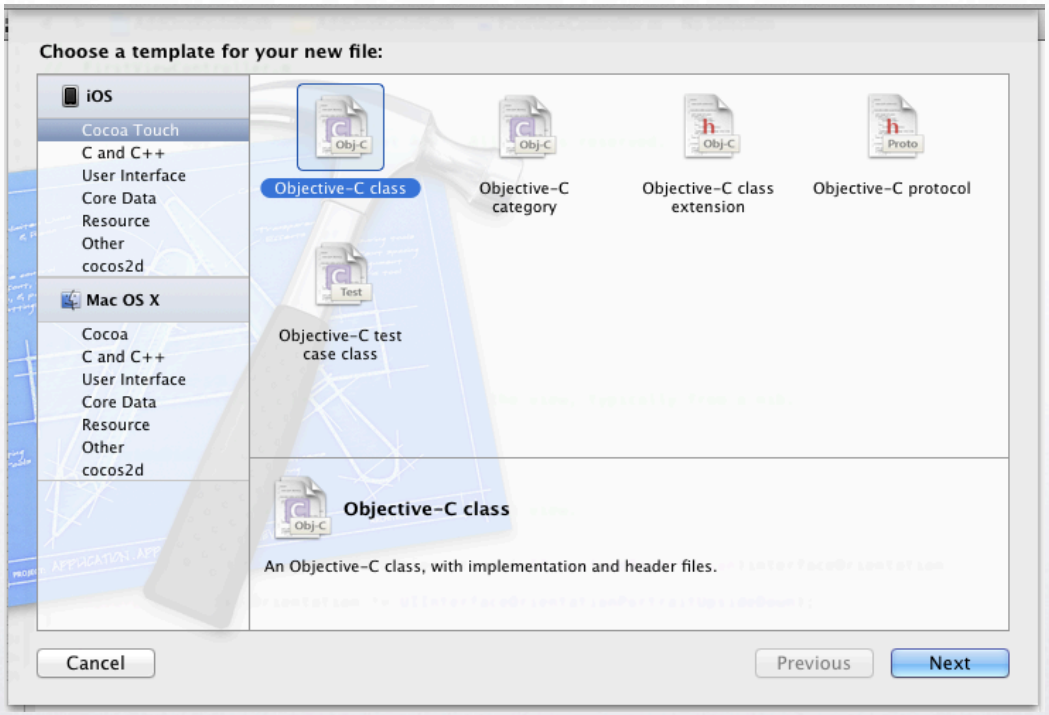


© May 2011, Michael Patrick Ellard

Adding a new Objective-C Class

At some point, you're going to want to know how to create a Tab Bar Controller with more than just the default two views. So let's add a third view.

First we'll create the view's controller class. Go to Xcode, and choose "New" submenu from the File menu, then choose the "File…" command. Choose "Objective-C class."



© May 2011, Michael Patrick Ellard

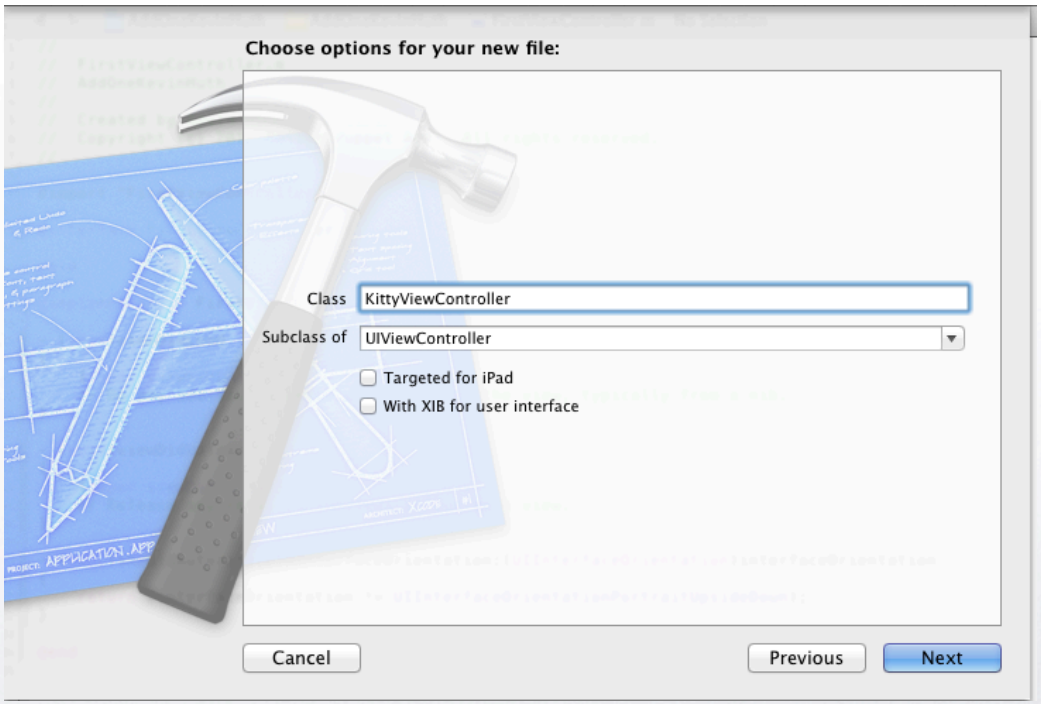
Choosing a Parent Class

On the next screen, we'll get to specify a parent class for our new View Controller. Use "UIViewController" in the Subclass of box.

Uncheck the "With XIB for user interface" box.

Name your new view controller "KittyViewController."

After you have added it to your project, choose "Build" from the "Product" Menu to make sure that your project builds without any issues.



© May 2011, Michael Patrick Ellard

Adding KittyViewController to our TabBar

Go to your MainStoryboard.storyboard.

Drag a View Controller from the Object Library over into your Storyboard.

Use the Identity Inspector to make the new view controller belong to the class KittyViewController.

Select the Tab Bar Controller, and use the Connections Inspector to see it's relationships.

Note that it has relationships with the First and Second View Controllers already.

Select the dark circle at the right side of these relationships and drag over to the KittyViewController that you've just created. This will add a new relationship for that view controller. Your KittyViewController is now part of the tab bar.

© May 2011, Michael Patrick Ellard

Updating the Tab Bar Image and Label

Finally, select the new view controller's "Tab Bar Item" and use the Attributes Inspector to set its Title to "Kitty" and its Image to "KittyIcon.png"

© May 2011, Michael Patrick Ellard

Updating KittyViewController.h

Make the following updates to KittyViewController.h:

```
@interface KittyViewController : UIViewController

@property (nonatomic, strong) IBOutlet UILabel *countLabel;

@property (nonatomic, strong) IBOutlet UIView *pettingView;

- (IBAction) addOne:(id)sender;
```

© May 2011, Michael Patrick Ellard

Updating KittyViewController.m

Make the following changes to your KittyViewController.m file:

```
#import "KittyViewController.h"
#import "ModelObject.h"

@implementation KittyViewController

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    ModelObject *myModel = [ModelObject sharedInstance];

    self.countLabel.text = [NSString stringWithFormat:@"Count is %d", myModel.currentCount];
}


- (IBAction) addOne:(id)sender
{
    ModelObject *myModel = [ModelObject sharedInstance];

    [myModel addOne];

    self.countLabel.text = [NSString stringWithFormat:@"Count is %d", myModel.currentCount];
}
```

© May 2011, Michael Patrick Ellard

Two more methods

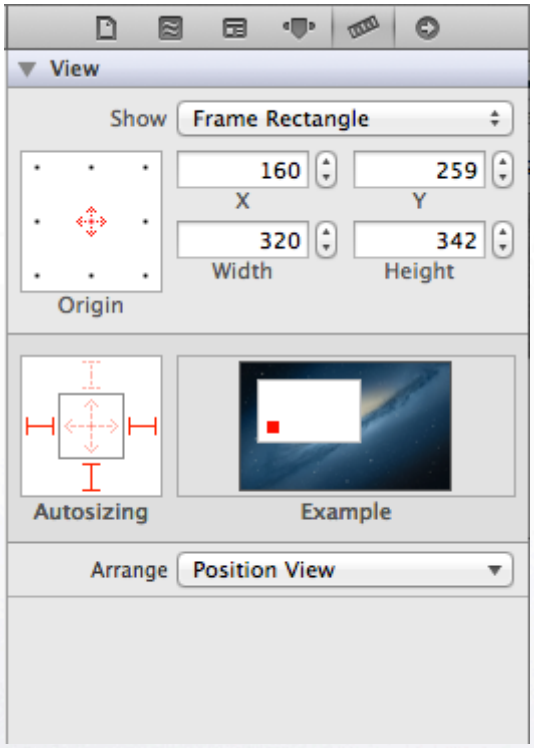
Next, add the following methods to the KittyViewController.m file.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    UISwipeGestureRecognizer *pettingGestureRecognizer = [[UISwipeGestureRecognizer alloc]
        initWithTarget:self action:@selector(addOne:)];

    [self.pettingView addGestureRecognizer:pettingGestureRecognizer];

    [pettingGestureRecognizer release];

    self.pettingView = nil;
}

-(void)dealloc
{
    self.countLabel = nil;

    [super dealloc];
}
```

When you are done with your updates, choose Build from the Product menu to incorporate these changes into your compiled code.

© May 2011, Michael Patrick Ellard

KittyView

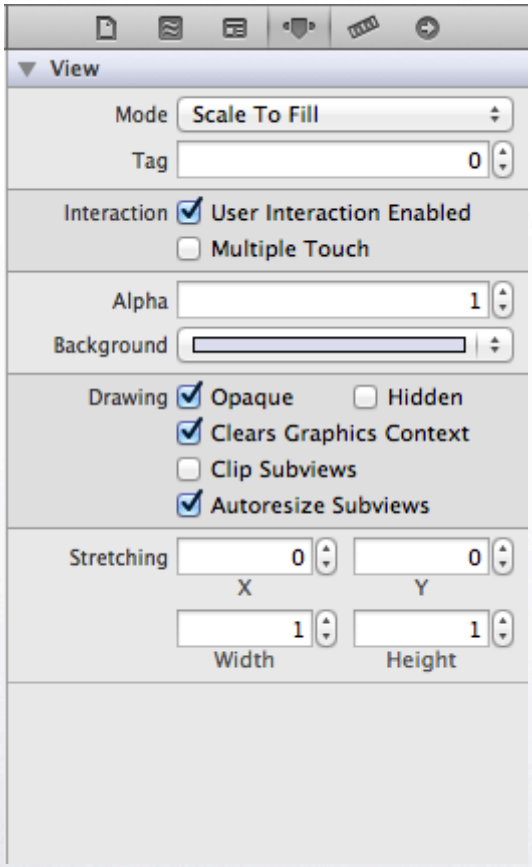


Now we're going to set up our KittyView. To do this:

- Click on the MainStoryboard.storyboard
- Add a UIImageView as we did with our other views.
- Use the Attributes inspector to set the image to Kittybackground.png
- Use the Size Inspector to make the UIImageView's
  - Width: 320
  - Height: 342
  - X: 160
  - Y:259
- Use the Autosizing properties so that this view will keep its size and be anchored to the bottom of the screen

© May 2011, Michael Patrick Ellard

Setting the Background Color

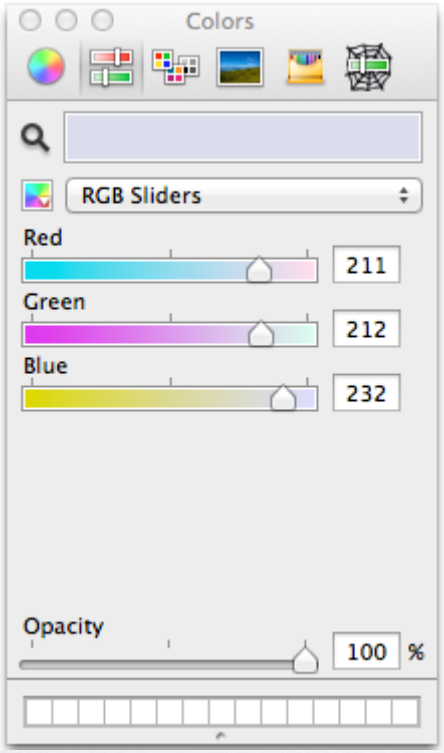


We want to change the background color of the main view for this story board to match the kitty graphic. To do this, select the main view for this screen from the Storyboard's structure pane.

Then open the Attributes inspector, and choose "Other" for the background color.

© May 2011, Michael Patrick Ellard

Choosing a Color Option



Choosing other will allow us to use a variety of color pickers.

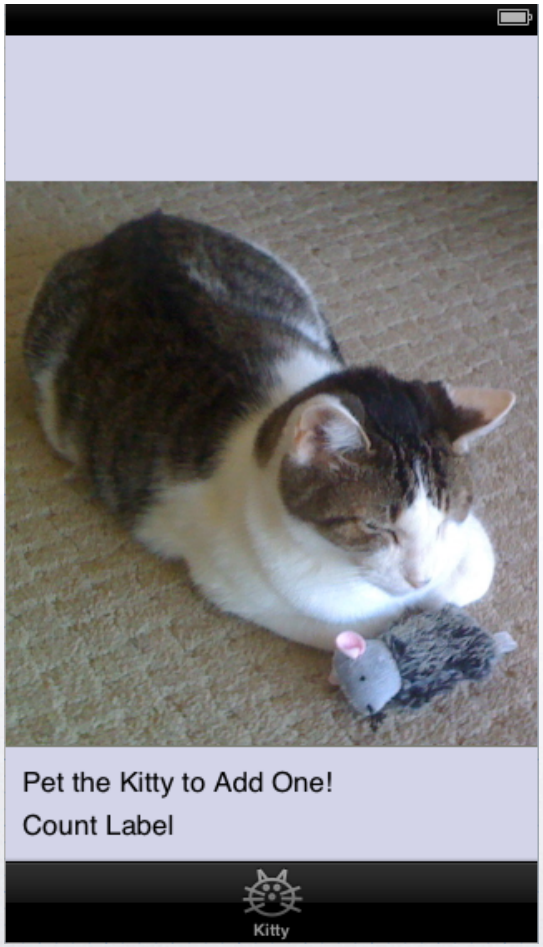
Choose the Slider option, then choose the RGB Sliders option.

For the colors, enter:

- Red: 211
- Green: 212
- Blue: 232

© May 2011, Michael Patrick Ellard

KittyView



Now we're going to set up our KittyView. To do this:

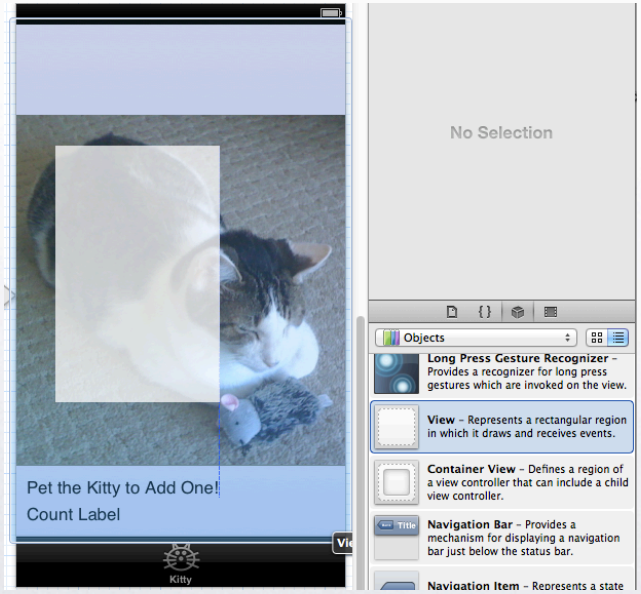
- Add a counter label.
- Add a second label that says, "Pet the Kitty to add one!"
- Make sure both labels are anchored to the bottom of the screen.
- Create a connection from the owning object for the Kitty View Controller to the counter label and use it to connect the countLabel outlet.

Your resulting view should look like the one to the left.

© May 2011, Michael Patrick Ellard

Hooking up our Gesture

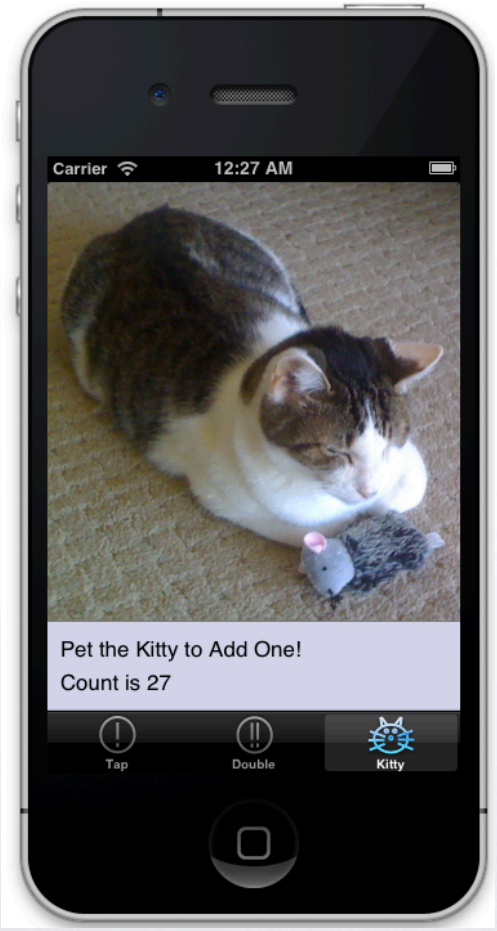
The final stop to hooking up our gesture is to drag a view over onto the area where our kitty is. Make the view nice and large - you want it to be big enough to be able to capture swiping gestures. After it is in place, make a connection between the KittyViewController and the view, and assign the pettingView property to the view. Finally use the Attributes Inspector to make the gesture-capturing view transparent by setting its Background to "Clear Color." After you've done this, click the run button.



Note: Don't make your gesture-capturing view transparent by making it "Hidden" or setting its Alpha to zero. If you do either of those things, the view will stop reacting to events and your swiping gestures won't be recognized.

© May 2011, Michael Patrick Ellard

Gesture Recognizer in Action



Congratulations! You should now have a working gesture recognizer.

The swipe recognizer works on left-to-right gestures. Other "swipes" or "pets" won't be recognized unless you add them.

Implementing a more sophisticated gesture recognizer is left as an exercise to the reader.

© May 2011, Michael Patrick Ellard

Model-View-Controller in Action

We used a Model-View-Controller design pattern in this project. Here are some of the benefits of doing so:

- We were able to add a second and third view controller without changing anything in our model or our first view controller.
- When we switch tabs, the view controller that we switch to automatically shows all of the changes to our data made by any of the other view controllers, even though none of the view controllers has a reference to the others.
- We could delete any of the view controllers from our project and the model and the other view controllers would continue to function flawlessly with no need for code changes.

© May 2011, Michael Patrick Ellard

Congratulations, You've Finished!

In this project you have:

- Learned how to create a project that uses UITabBar for navigation
- Gotten experience working with an MVC app
- Successfully used a Singleton object
- Incorporated legacy C code into an Objective-C project
- Practiced using Xcode to create a multi-view App
- Gotten experience with IBOutlet
- Worked with the new Storyboard features of iOS 5
- Created screen layouts that automatically adjust for different screen sizes
- Begun to explore the different gesture recognizers available to you under iOS

Great work! Well done!

© May 2011, Michael Patrick Ellard