Intermediate iOS

CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 17 — 10/23/2012

Goals of the Lecture

- Learn more about iOS
  - In particular, the ins and outs of view controllers and storyboards
  - Plus
    - Navigation Controllers
    - Tab Bar Controllers
    - Table View Controllers
      - All of these allow us to explore storyboards with multiple view controllers

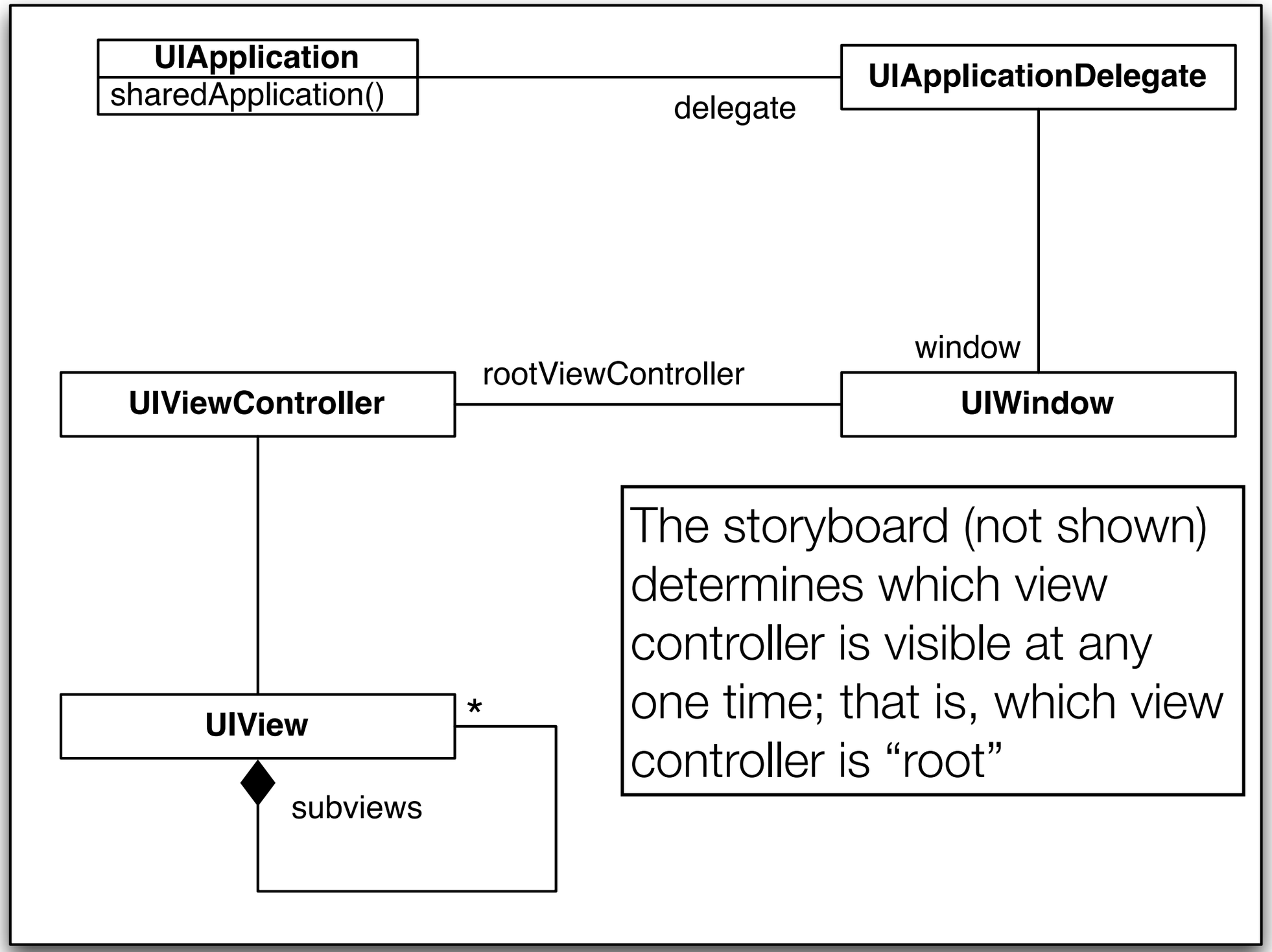
Credit for Icons

- For the Tab Bar Controller example, I make use of a free set of icons developed by APP-BITS
  - APP-BITS website: <http://app-bits.com/>
  - Free Icons: <http://app-bits.com/free-icons.html>
- I've included the entire distribution inside of the sample code for Lecture 17

iOS Fundamentals (I)

- Each iOS application has
    - one application delegate
    - one window
    - one or more view controllers
        - each view controller has one view that typically has many sub-views arranged in a tree structure
            - e.g. views contain panels contain lists contain items…
- one storyboard (if present) that specifies the interconnections of the application's view controllers

iOS Application Architecture



iOS Fundamentals (II)

- A window has a “root” view controller
  - It’s view fills the entire window
  - Some view controllers act as “container controllers”
    - They have a set of view controllers that they manipulate in various ways
      - A **tab bar controller** will have one view controller per tab
      - A **navigation controller** will have a set of view controllers that it displays using a stack model
        - When you transition in a navigation controller, you are either pushing a new view controller onto the stack (which then becomes visible) or you are popping a view controller off the stack revealing the one beneath it

iOS Fundamentals (III)

- At other times, we may switch the “root” view controller entirely
  - the new view is displayed and the previous view controller (and its view) is deallocated
- In all of these cases, storyboards can handle most of the transitions for us
- View controllers sit between our model objects and their views
  - They access data and make sure the data appears correctly in the view
  - They receive events from widgets and respond appropriately
- Let’s see some examples!

Multiple View Application

- Let's see an example of an iOS app with multiple view controllers
  - Each view controller will have a view that
    - has a distinct background color
    - has a label that shows how many times this view has been visited
    - contains buttons that navigate among the other view controllers
- We will create a Single View application called MultipleViews
  - This provides us with the source code for one view controller and a storyboard that is configured to load it as the root view controller
  - We will then add two new view controllers to the application and use segues to transition between the controllers

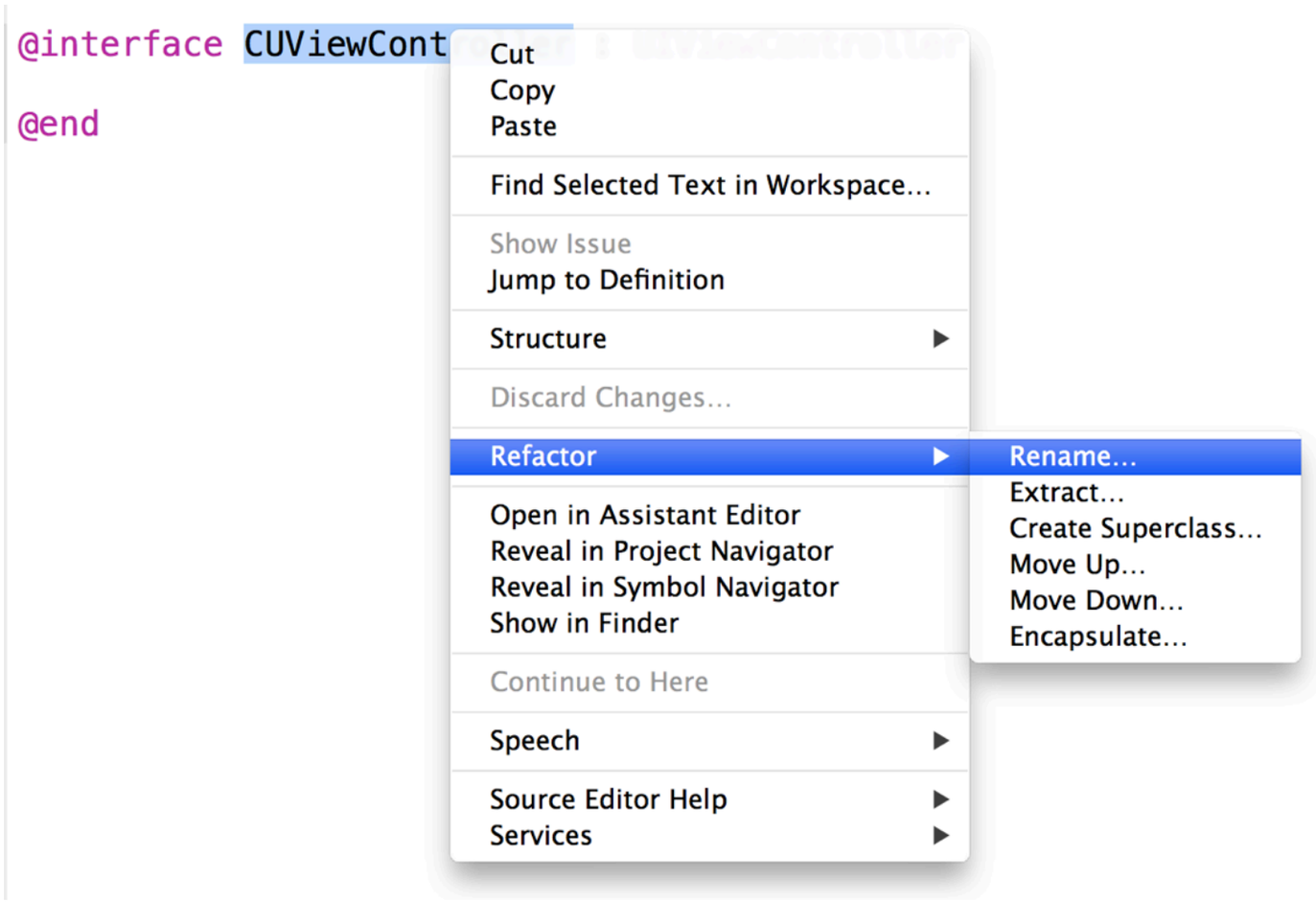
Step 1: Rename First View Controller

- We want to create three view controllers
  - RedViewController
  - GreenViewController
  - BlueViewController

- The default view controller, however, was called CUViewController
  - I want to rename it to be RedViewController but renaming can be tricky
    - The current storyboard points at CUViewController; if I just rename the class, I might mess up the connections that the template created
    - This calls for refactoring

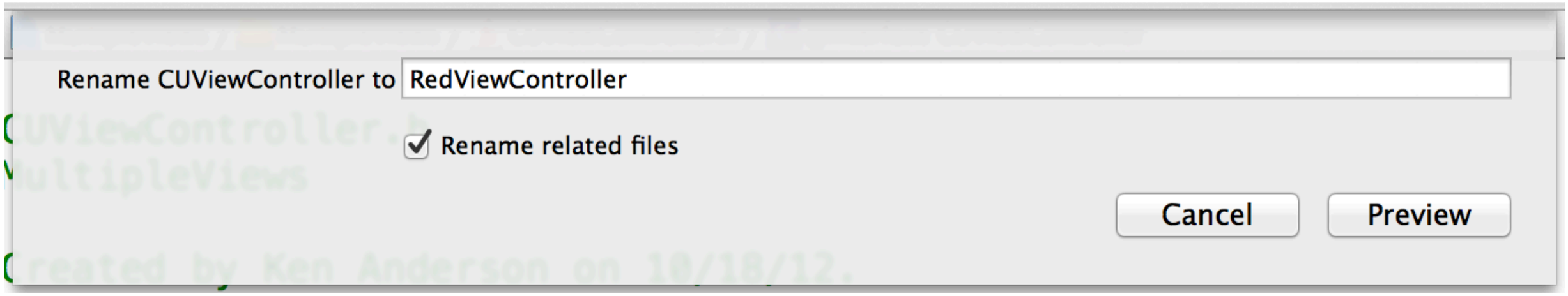
Refactor ⇒ Rename… (I)

- Click on CUViewController.h and “control click” or right click the name of the class
  - In the subsequent pop-up menu, select Refactor ⇒ Rename…



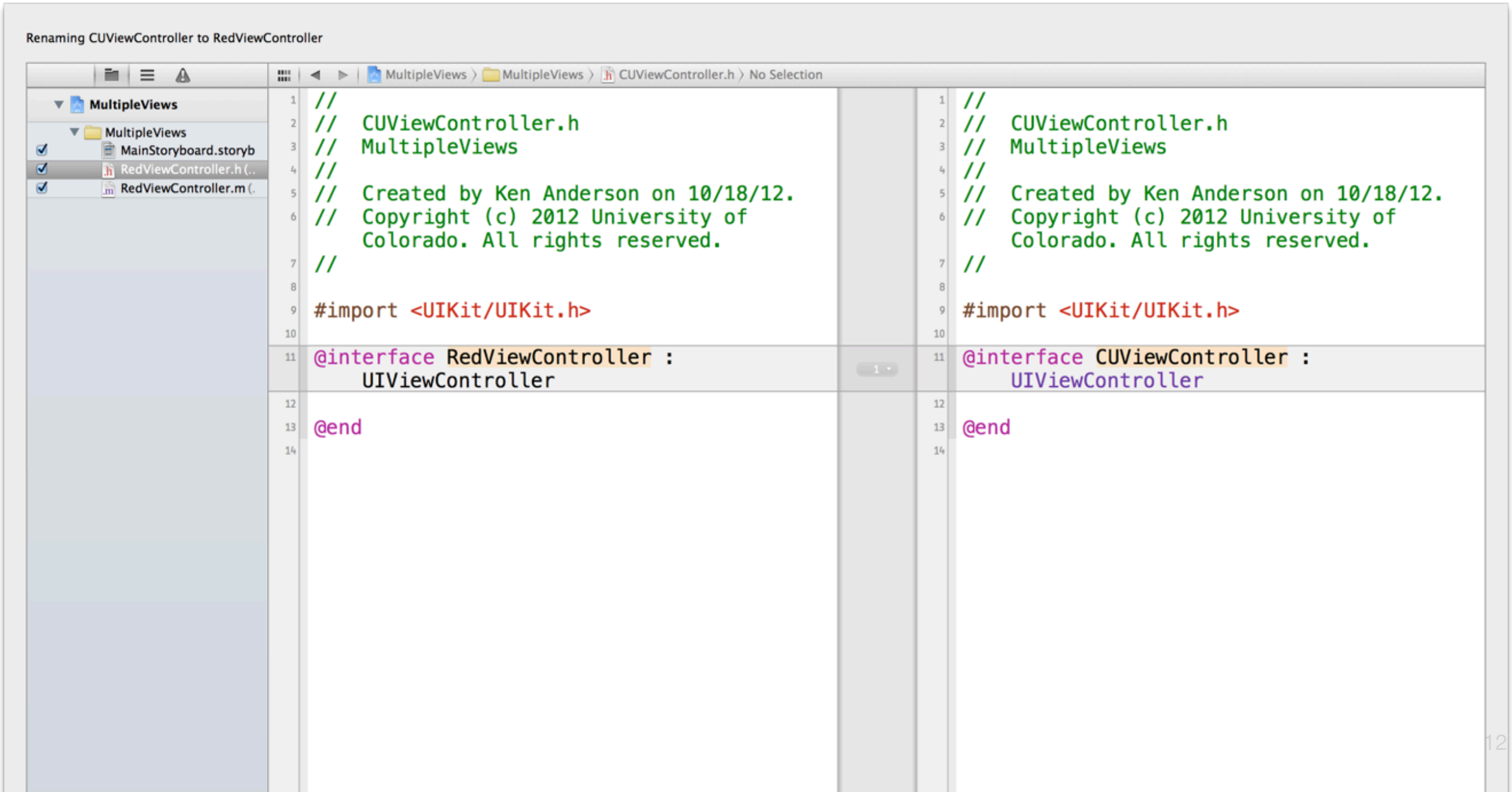
Refactor ⇒ Rename… (II)

- This brings up a dialog that lets you specify the new name of the class
  - Be sure to select “Rename related files” and then click Preview



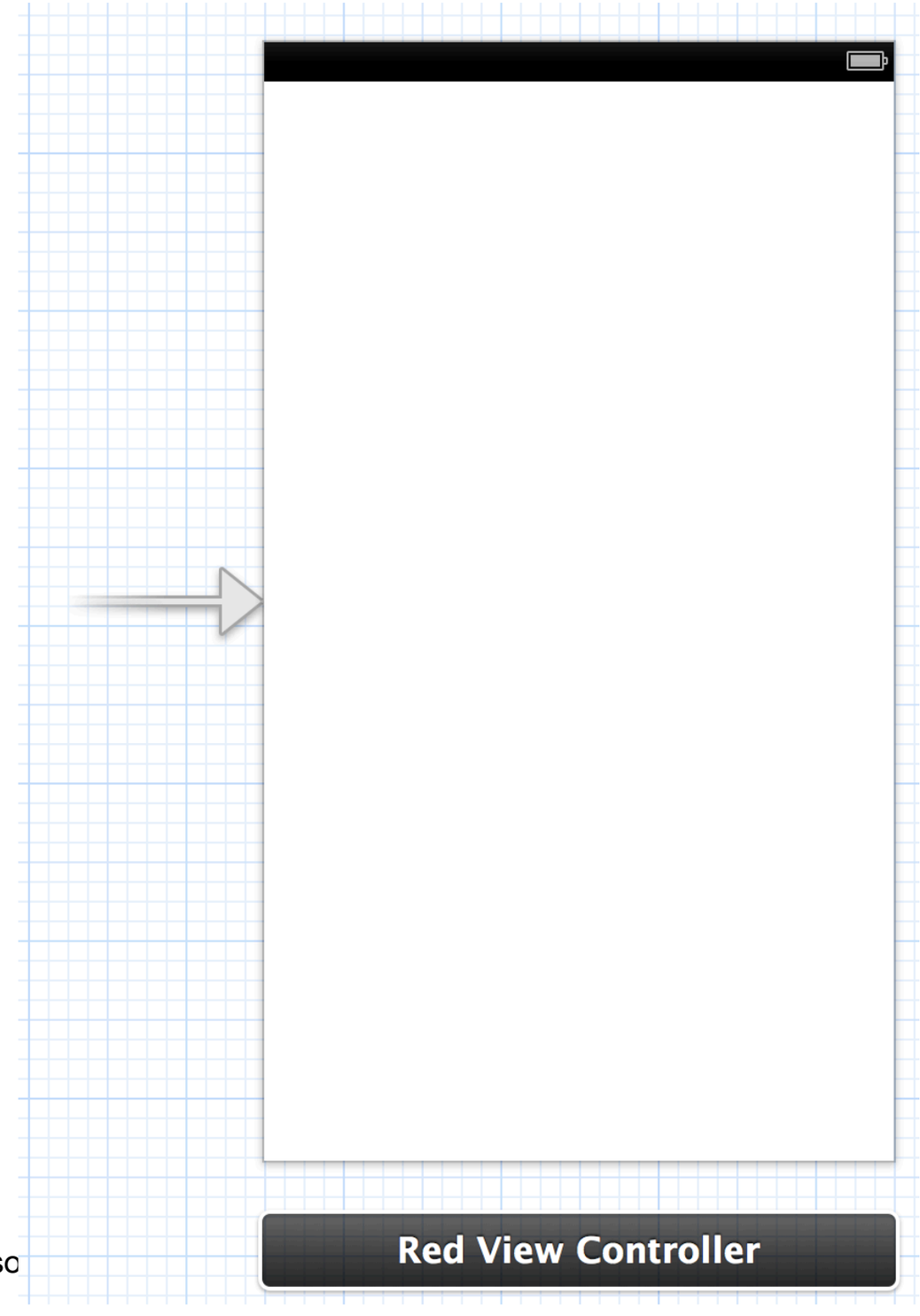
Refactor ⇒ Rename… (III)

- As you can see, the refactoring renames the class, the class files, and the storyboard!

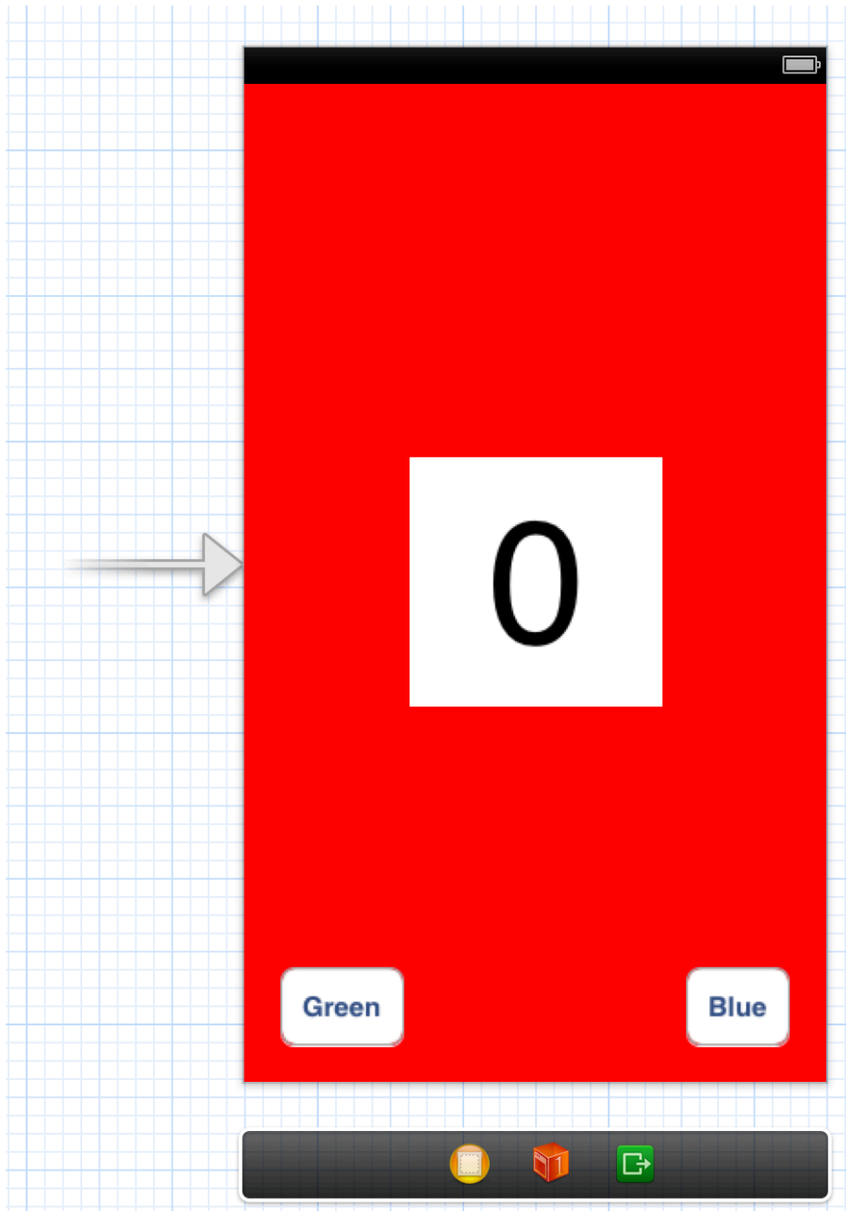


Refactor ⇒ Rename… (IV)

- Now, when I select the storyboard, it has been updated to use the newly renamed view controller
- We will now configure this view by
  - changing its background color to red
  - adding a label with a big font for the “view count”
  - add two buttons: green and blue
  - add properties to store view counts and one to connect to the label



Finished Interface



Finished Header File

```
9  #import <UIKit/UIKit.h>
10
11 @interface RedViewController : UIViewController
12
13 @property (nonatomic) int redCount;
14 @property (nonatomic) int greenCount;
15 @property (nonatomic) int blueCount;
16
17 @property (weak, nonatomic) IBOutlet UILabel *countLabel;
18
19 @end
```

- The count properties stores the number of times we've visited each of the three view controllers. These properties are automatically set to zero.
- The countLabel property points to the label and will let us display the value of the viewCount property

When do we update the value?

- Each time the view controller is displayed, we want to
  - increment viewCount
  - update the label to display the latest count
- We need to make sure we do this at an appropriate time

- We can't use `viewDidLoad` like we did in lecture 16 when we initialized the "text to speech" engine since that method is only called once
- Instead, we want to be notified just before the view controller's view is displayed, which can happen multiple times during the lifetime of a view controller as it gets hidden and redisplayed
  - the method for that life cycle event is `viewWillAppear:`

First version of the .m File

```
15 @implementation RedViewController
16
17 - (void)viewWillAppear:(BOOL)animated {
18     [super viewWillAppear:animated];
19     self.countLabel.text = [NSString stringWithFormat:@"%d", ++self.redCount];
20 }
21
22 @end
```

- Our implementation of viewWillAppear: does two things
  - It calls viewWillAppear: on our super class: UIViewController
    - It does this to have UIViewController handle any animation that needs to occur to make this view appear
  - It then uses C's ++ operator to first update the value of self.redCount, then it converts that value to a string, and passes that string to the label for display; All in one line of code!

Create the Other View Controllers (I)

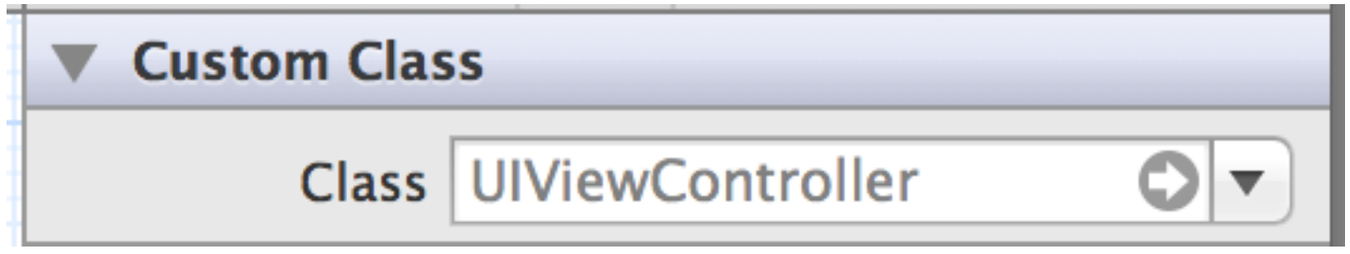
- Select the storyboard and bring up the Object Library
  - Search for a View Controller object and drag it out onto the storyboard
  - Do that one more time
  - You now have a storyboard with three view controllers
    - One points to the RedViewController
    - The two new ones just point to generic UIViewController instances
      - We'll change that in a moment

Create the Other View Controllers (II)

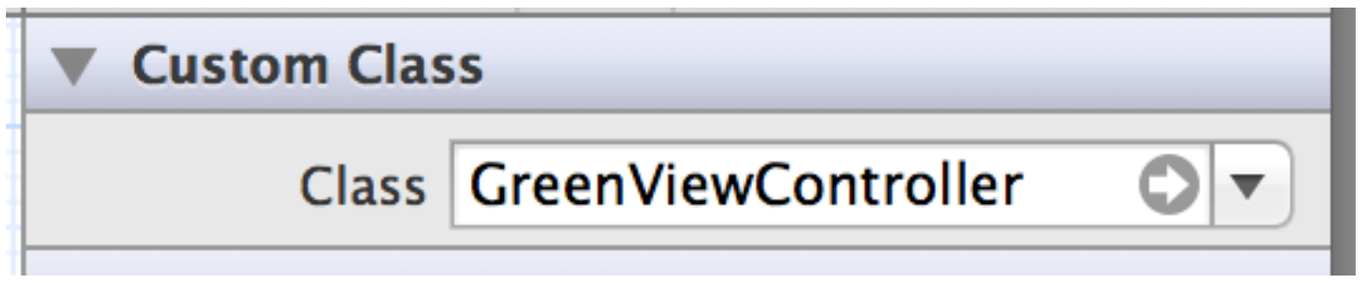
- Use File ⇒ New ⇒ File… to add a new UIViewController subclass called GreenViewController
  - This will result in GreenViewController.h and GreenViewController.m to be added to our project
- Do this one more time and call the third view controller BlueViewController
  - Note: you can delete all of the template code that appears in GreenViewController.m and BlueViewController.m
  - Not the class definition, just the methods that appear within!
- These source code files are NOT currently connected to the new view controllers in the storyboard
  - Let's fix that

Create the Other View Controllers (III)

- To attach the new view controllers in the storyboard with the new source code files we need to use the Identity Inspector (⌥⌘3)
  - Select one of the new view controllers in the storyboard
  - When you look at the Identity Inspector it will reveal that the class of this object is UIViewController; Change the Class to GreenViewController



Before

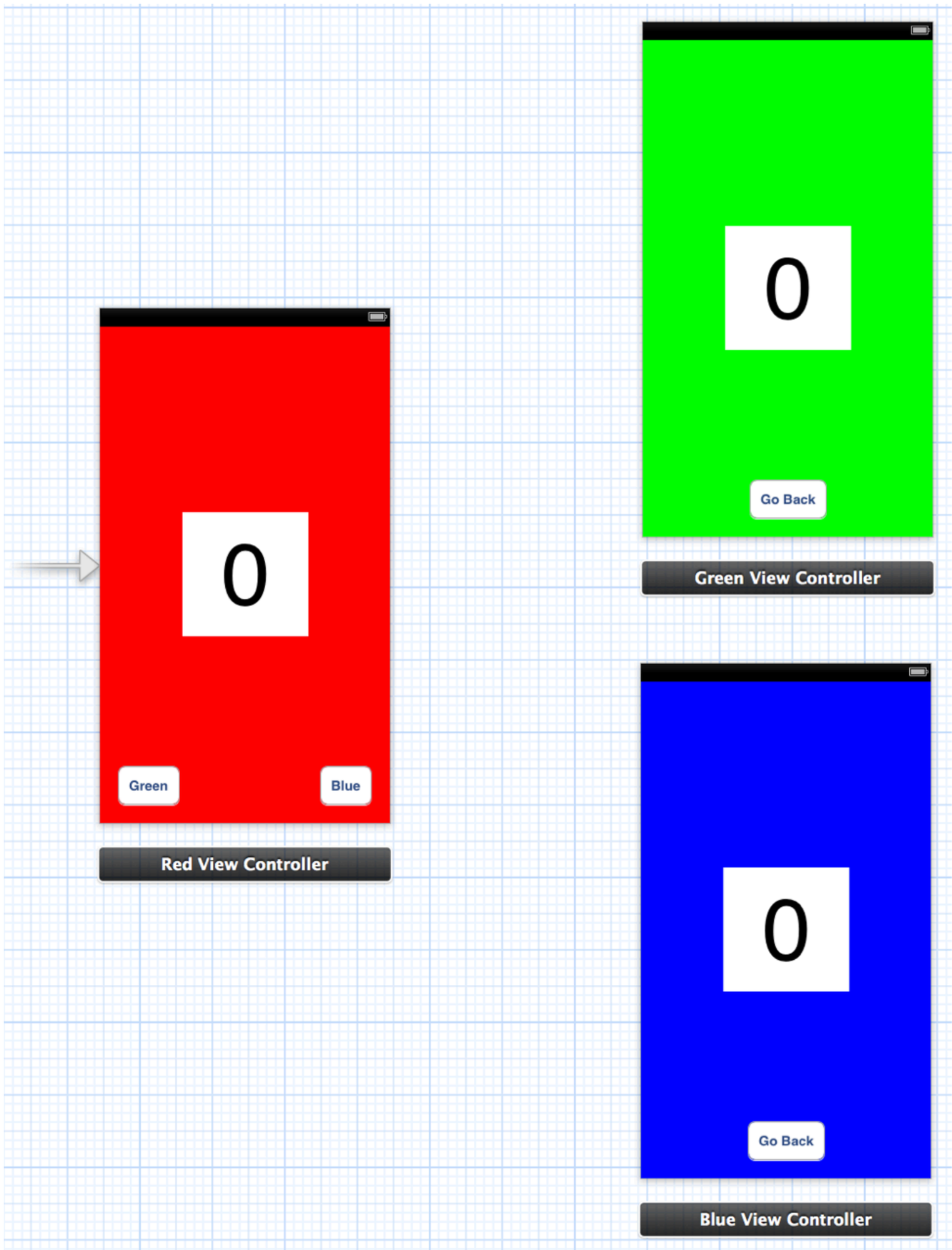


After

- Select the other view controller and change its class to BlueViewController

Configure View Controllers and Code

- Now, we need to apply similar changes to our new view controllers
    - We need to add buttons and labels, change background colors, create properties, and implement viewWillAppear:
- Note: segue transitions follow “stack conventions”
    - As such, the green and blue controllers have “Go Back” buttons
        - More on this in a minute
- The storyboard will look similar to the image on the next slide



Verify Configuration is Correct

- We can now verify that our configuration of these three view controllers is correct by running the application three times
  - Each time, we take the arrow in the storyboard that points to the first view controller
    - and point it at a different view controller: first red, then green, then blue
      - Each time that you run the app, you should see the selected view controller and its label should read “1” for the RedViewController and “0” for the other two
- The buttons don’t currently do anything
  - We’ll fix that next

Segues (I)

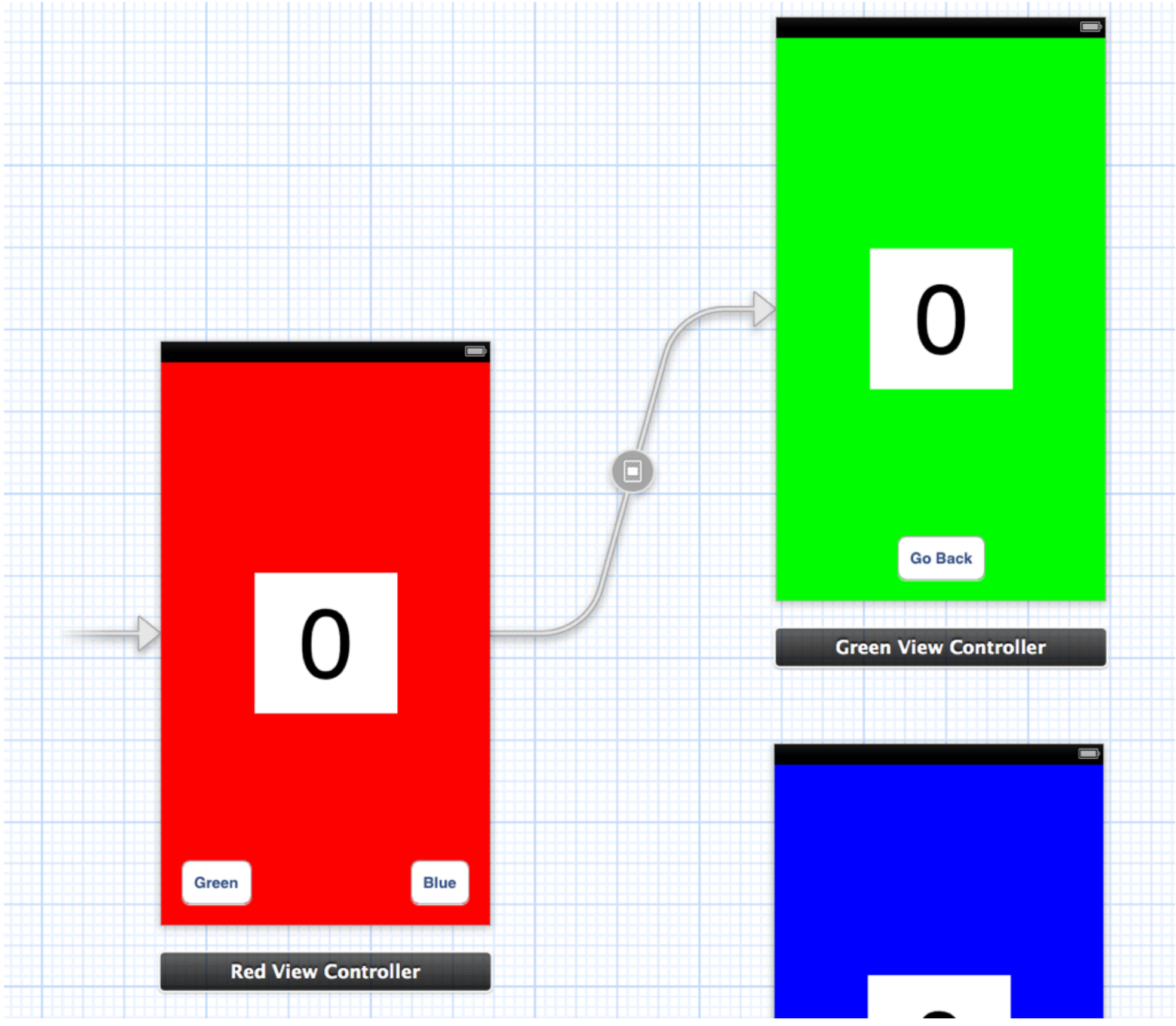
- A segue specifies a transition from one view controller to another
    - These transitions form a stack
        - You have your initial view controller and its view
        - If you follow a segue to a second view controller
            - it gets pushed on the stack
            - and displayed
- It can then
    - either dismiss itself and return to the previous view controller
    - or follow another segue to push a third view controller onto the stack

Segues (II)

- As a result, our app is going to do the following
  - Display the red view controller as the root view controller
  - When the Green button is pushed, a segue will transition to the Green View Controller
    - We push it's Go Back button to return to the Red View Controller
  - We do a similar thing when the Blue button is pushed, this time transitioning to the Blue View Controller
- The red view controller will keep track of how many times we have visited each view controller and pass that value to the green and blue controllers for display
  - As a result, their viewWillAppear methods simply display their viewCount properties; we do not increment the value like we did for the red controller

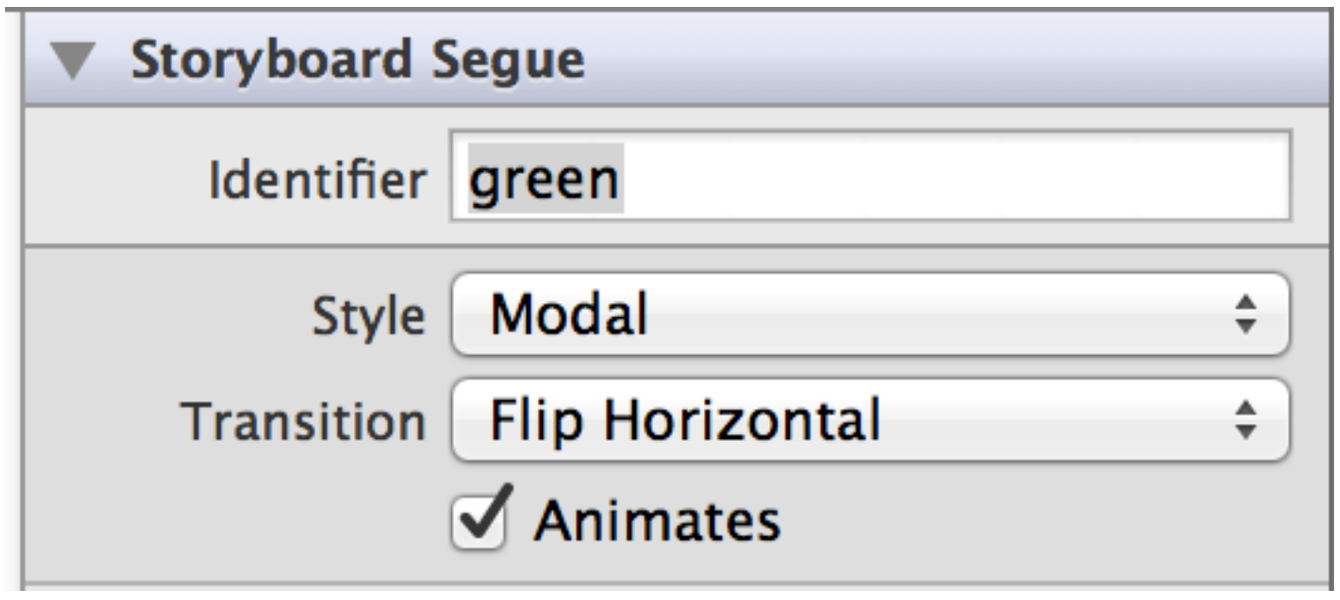
Creating our First Segue

- Select the storyboard and select the RedViewController
  - Control click on its Green button and drag to the GreenViewController
  - Let go and a menu asks what type of segue to create
  - Select Modal; Our segue is created (see next slide)



Configure First Segue

- Select the segue and bring up the Attributes Inspector (⌥⌘4)
  - Use this dialog to give this segue the identifier “green”
  - Select the Flip Horizontal transition



- Run the app and click the Green button. Fun!
  - No lines of code! However, the Green View Controller’s label said “0”; we’ll fix that after we create the second transition

Create and Configure Second Segue

- Control-Click on the Blue button and drag to the BlueViewController
- Select “modal” to create the second segue
- Configure this segue to have an identifier of “blue” and use the transition “Cover Vertical”
- Save and Run the Application; Click the Blue button and enjoy!

- However, the Go Back buttons do not work
  - Let’s fix that

Create handleGoBack actions

- Use the technique from Lecture 16 to create handleGoBack methods in the Blue and Green view controllers
  - In those methods, we simply call the following method
    - dismissViewControllerAnimated:completion:
    - Like this

```
22  - (IBAction)handleGoBack:(UIButton *)sender {
23      [self dismissViewControllerAnimated:YES completion:NULL];
24  }
```

- With this in place, we can now move back and forth between all three view controllers
  - One last problem: the count for Blue and Green is always “0”

Communicating Between View Controllers (I)

- In order to get the blue and green view controllers to display a count other than zero, we need to have the red view controller tell them what to display
- To do this, we must implement a method that
  - lets the red view controller configure the destination view controller
    - after it's been created
    - but before it's been displayed on the screen
- That method is called `prepareForSegue:sender:`
  - This method passes in a segue object that we can use to retrieve the destination view controller; we can then configure it
    - We use our segue identifiers to figure out what to do (see next slide)

Implementation of prepareForSegue:sender:

```
25 - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
26     if ([segue.identifier isEqualToString:@"green"]) {
27         GreenViewController *vc = (GreenViewController *)segue.destinationViewController;
28         vc.viewCount = ++self.greenCount;
29     } else if ([segue.identifier isEqualToString:@"blue"]) {
30         BlueViewController *vc = (BlueViewController *)segue.destinationViewController;
31         vc.viewCount = ++self.blueCount;
32     }
33 }
```

- This method appears inside of RedViewController
- It checks the identifier property of the segue parameter to determine if we are transitioning to the green view controller or the blue view controller
- It then type casts the destination view controller property to the appropriate type, increments the appropriate count property and assigns that value to the destination view controller's viewCount property

We're done!

- With that, this application is complete
    - You can now transition between the various views and see their counts update
        - In the past, you would have had to code these transitions manually
        - Now the storyboard takes care of it for you while still allowing you to pass information between view controllers via prepareForSegue:
- If you "quit" the app, you'll see that its process simply moves to the background; that's because if you "relaunch" the app, you'll see that the counts have not been reset to zero

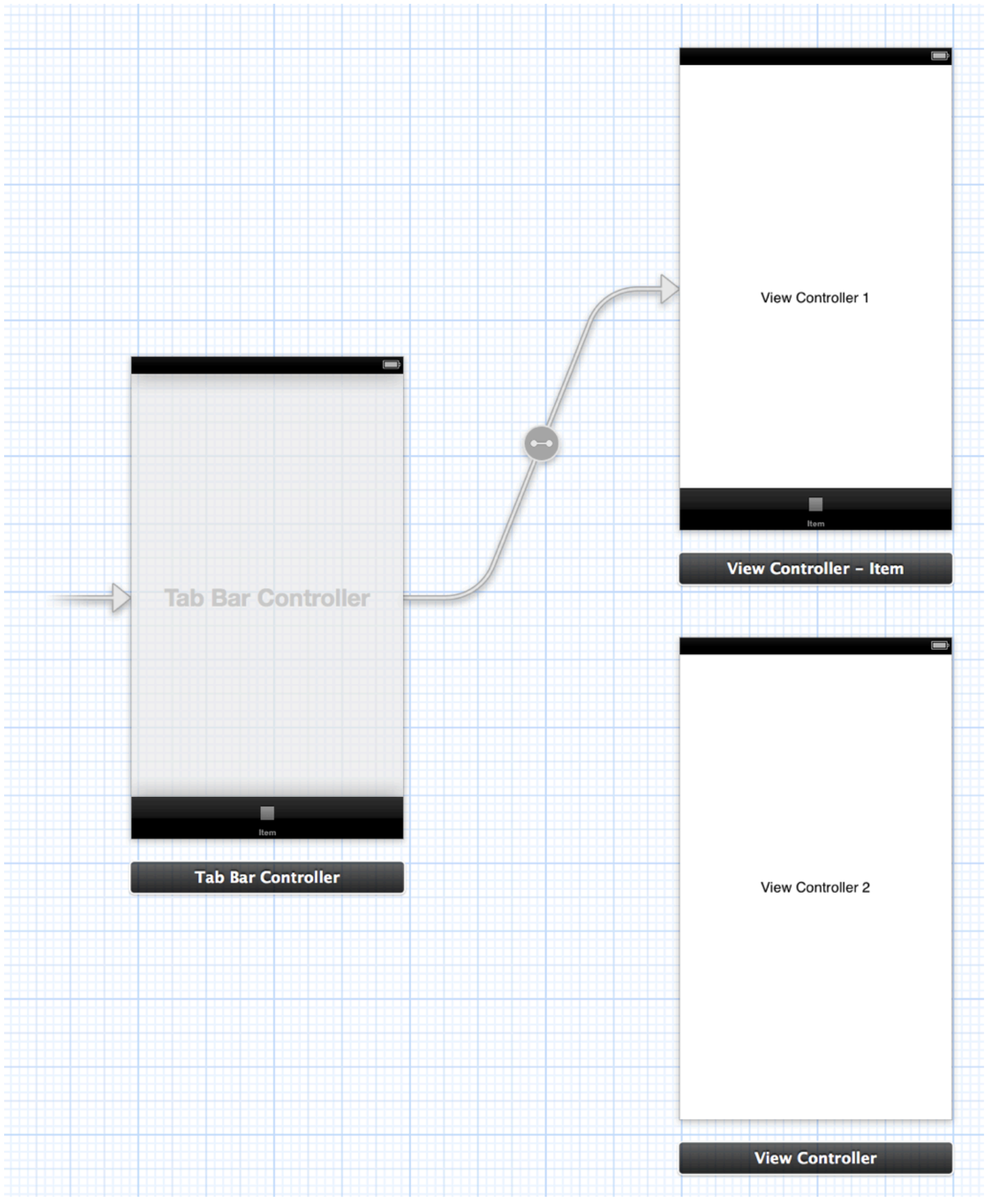
Tab Bar Application

- Let's see an example of using a storyboard to create a tab bar application
  - A tab bar view controller is an example of a container controller
    - It manages one view controller per tab and allows the user to switch between them
- We're going to go bare bones with this application and just create two tabs that display simple view controllers that each have a label that helps to distinguish one from the other

Step One: Basic Set-Up

- Create a Single View App and call it TabBar
- Select the storyboard and add a new view controller
- Put a label on the first view controller's view that says "View Controller 1"
- Put a label on the second view controller's view that says "View Controller 2"

- Select the first view controller and invoke the following menu command
  - Editor ⇒ Embed In ⇒ Tab Bar Controller
- A Tab Bar Controller is added and the first view controller has been automatically configured to be its first tab (!)
  - Your storyboard should now look similar to the next slide



Step Two: Create Second Tab

- Control-Click on the Tab Bar Controller and drag to the second view controller
- Let go and a menu appears with more options than in our previous example
  - In particular, there's a new section called "Relationship Segue" and a choice under that which says "view controllers"
- Select the "view controllers" choice
  - This essentially says that we'd like the second view controller to be one of the view controllers managed by the tab bar controller
- Presto! Just like that we have a Tab Bar app with two tabs
  - If you run it, you can click on the two tabs to see your two view controllers in action

Step Three: Configure the Tabs

- The application works but the tabs themselves look horrible
  - Let's provide them with names and cool icons
- To do this, pick two icons from the APP BITS folder and drag them into your XCode project
  - Using the same technique that we used to add the "text to speech" framework to our XCode project
  - I selected pacman@2x.png and scales@2x.png
- Now that the icons have been added to the project, click on each view controller and near the bottom click on its tab icon
  - Give each tab a name and an image using the Attributes inspector

Step Four: There is No Step Four

- We’re done!
    - As you can see, it is straightforward to create a tab bar application
        - Create a set of view controllers
        - Select one and embed it in a tab bar controller
        - Control drag from the tab bar controller to each view controller
        - Configure their tab icons and names
        - Work on the individual view controllers until the application is complete

The Social

- Let’s create an a contact app that focuses on social media
  - We’ll create a (very) simple model
  - And then progressively add (not necessarily in this order)
    - A view that displays and edits this information
    - A table view that lists multiple contacts
    - A nav controller that lets us move between these two views
- We’ll call this app “The Social” and we’ll start (as usual) with a single view app that uses storyboards and automatic reference counting

The Model (I)

- Our model class (ContactInfo) is going to be an objective-c class with the following properties
  - name - the name of a contact
  - twitter - the contact's twitter handle
  - facebook - the contact's facebook page
  - e-mail - the contact's boring old e-mail address
- To handle having multiple contacts, we'll simply have one of our view controllers store them in an NSMutableArray

The Model (II)

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

@interface ContactInfo : NSObject

@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *twitter;
@property (nonatomic, copy) NSString *facebook;
@property (nonatomic, copy) NSString *email;

@end

```

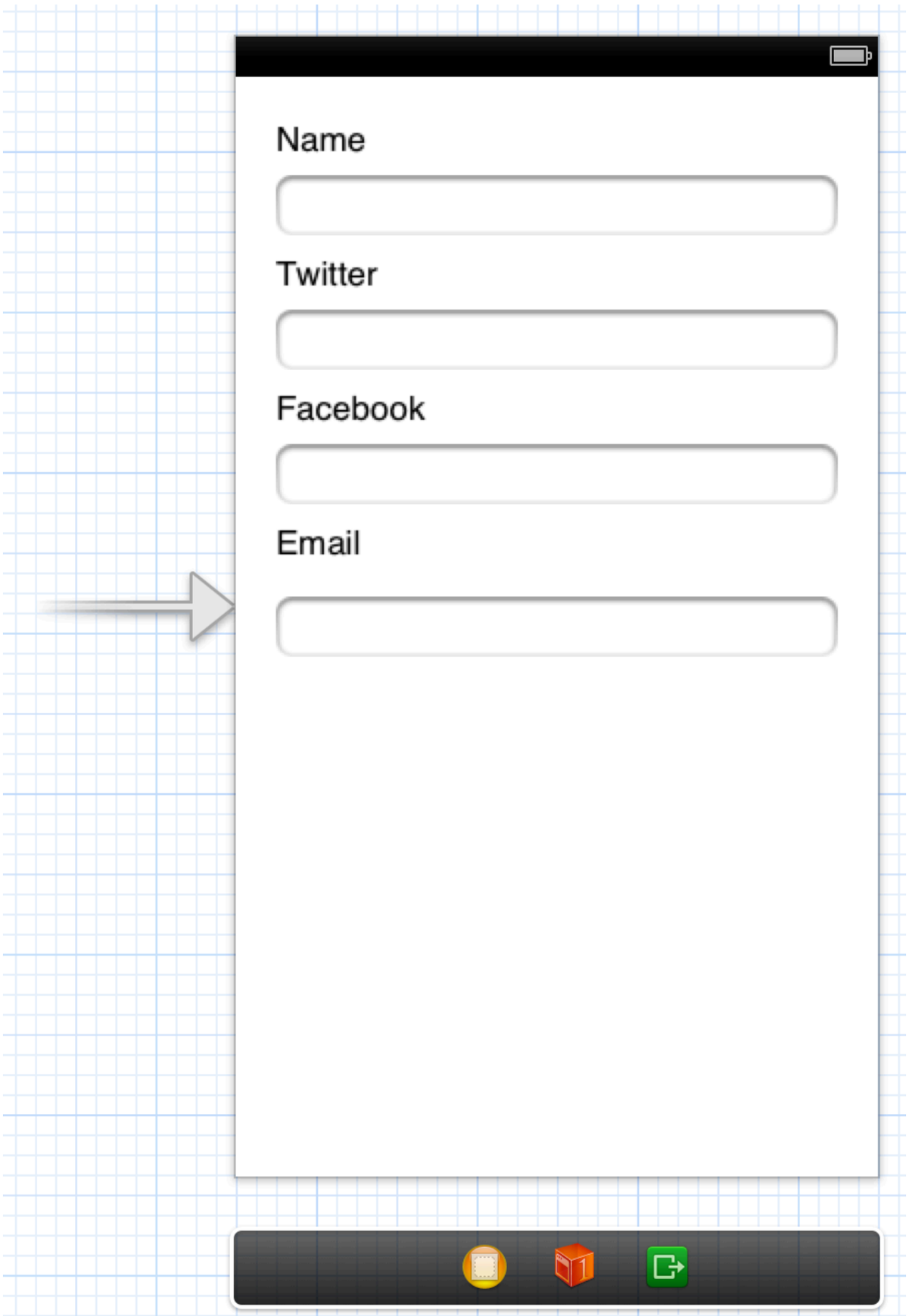
- This is what a “dumb data holder” looks like in Objective-C; for this app, we’re just going to store values in our model object; we can leave the .m file for ContactInfo completely blank

The Contact Detail View

- We need a view that will let us display and edit the information of a single instance of ContactInfo
- We will create a ContactDetailViewController
    - It will contain labels and textfields for each of the four properties of ContactInfo
    - It will have a ContactInfo property that it will use to retrieve the information that it should display
    - It will use the “button in the background” trick that we learned in lecture 16 to make the keyboard go away when we have finished editing a field
    - We will not currently attempt to “save” our edits; we will handle that later

Creating ContactDetailViewController

- We will now perform the following steps
  - Use the “rename class” refactoring to change the name of the automatically created CUViewController to ContactDetailViewController
  - Add labels and textfields to our view in Interface Builder
    - Add properties for each textfield using the “assistant editor” technique
- Add a property called info to the .h file of type ContactInfo*
- Add an implementation of viewWillLoad to the .m file that pulls values out of the info property and places them into the widgets
- Create an instance of ContactInfo in CUAppDelegate and pass it to our root view controller (our ContactDetailViewController) so it has something to display



Final Interface for ContactDetailViewController

Couple of Details

In attributes inspector, I configured the keyboard for Facebook to be of type URL and the keyboard for Email to be of type Email Address

Also, note the invisible button that is positioned behind the labels and textfields

Header File for ContactDetailViewController

```
9  #import <UIKit/UIKit.h>
10
11 @class ContactInfo;
12
13 @interface ContactDetailViewController : UIViewController
14
15 @property (nonatomic, strong) ContactInfo *info;
16
17 @end
```

- All we need is a property to store a single instance of ContactInfo*
  - Note use of @class to tell the compiler that ContactInfo is a class defined somewhere else, without having to #import the whole class definition

Implementation File for ContactDetailViewController

```objc
#import "ContactDetailViewController.h"

#import "ContactInfo.h"

@interface ContactDetailViewController ()

@property (weak, nonatomic) IBOutlet UITextField *nameField;
@property (weak, nonatomic) IBOutlet UITextField *twitterField;
@property (weak, nonatomic) IBOutlet UITextField *facebookField;
@property (weak, nonatomic) IBOutlet UITextField *emailField;

@end

@implementation ContactDetailViewController

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    self.nameField.text = self.info.name;
    self.twitterField.text = self.info.twitter;
    self.facebookField.text = self.info.facebook;
    self.emailField.text = self.info.email;
}

- (IBAction)dismissKeyboard:(UIButton *)sender {
    [self.view endEditing:YES];
}

@end
```

To initialize view, we simply copy values from our info property

Because we have multiple text fields, we use the endEditing: method to dismiss the keyboard

Implementation File for CUAppDelegate (for now)

```
#import "CUAppDelegate.h"

#import "ContactInfo.h"
#import "ContactDetailViewController.h"

@implementation CUAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)
    launchOptions {

    ContactInfo *info = [[ContactInfo alloc] init];
    info.name = @"Steve Martin";
    info.twitter = @"SteveMartinToGo";
    info.facebook = @"http://www.facebook.com/SteveMartinofficial";
    info.email = @"Steve.Martin@example.com";

    ContactDetailViewController *vc = (ContactDetailViewController *)self.window.rootViewController;
    vc.info = info;

    return YES;
}

@end
```

When didFinishLaunchingWithOptions is invoked, we know that 1) our view controller has been loaded and set as the rootViewController on our window and 2) the user interface has NOT appeared on screen

So, we create an instance of ContactInfo, retrieve our view controller, and set its info property; We can now run the program and see our info displayed

The ContactTableViewController

- We now need a view controller that presents a list of contacts
  - In iOS a TableViewController is the most common way to present a list
    - It's a bit of a misnomer because TableViewControllers can only handle a single column of information
      - However, you can have as many rows as you want!
- TableViews rely on two roles
  - a data source (to provide the data it needs)
  - a delegate (to handle events on the table view)
- These two roles are typically played by a single object: the TableViewController

Creating ContactTableViewController (I)

- In our storyboard, drag out a table view controller object (see next slide)
  - These are interesting beasts
    - A table view controller contains a table view which contain table view cells
    - In Interface Builder, you use the table view cells to define “prototype” rows that your application can then use to populate your table view at runtime
    - if you add additional cells in Interface Builder you’re saying that your table view can display more than one type of row
      - We’re going to use just one prototype cell and we’re going to configure it to use the “Subtitle” style and have an identifier of “Cell”

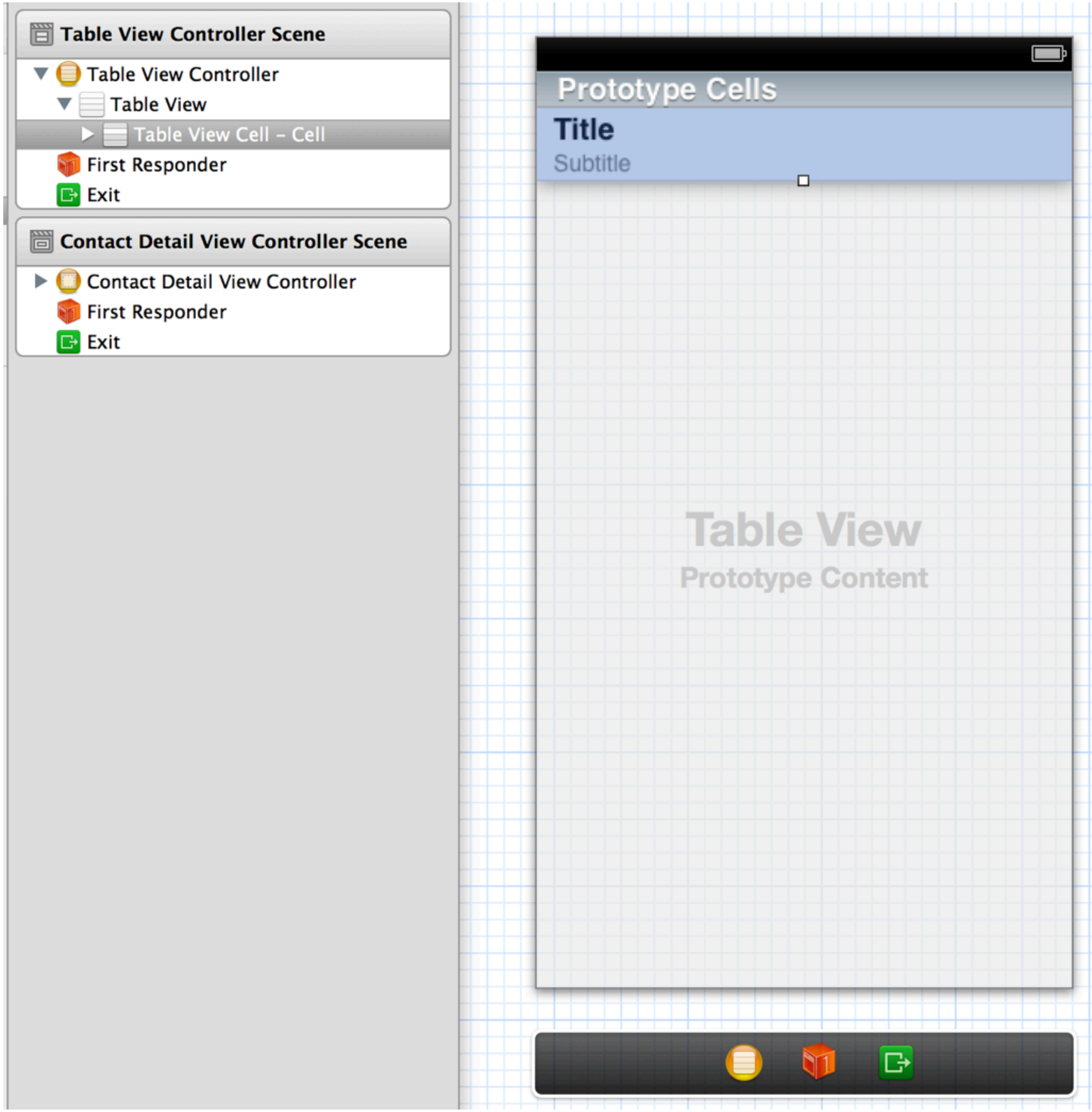


Table View Controller Scene
Table View Controller
Table View
Table View Cell – Cell
First Responder
Exit
Contact Detail View Controller Scene
Contact Detail View Controller
First Responder
Exit
Prototype Cells
Title
Subtitle
Table View
Prototype Content

Creating ContactTableViewController (II)

- Now, we add a new file to our project named ContactTableViewController
  - It will be a subclass of UITableViewController
- Once we have this file, we need to switch back to the storyboard, select our table view controller object and use the Identity Inspector to change its type to ContactTableViewController
- We now drag the arrow in our storyboard that indicates which view controller is “root” and point it at the table view controller
  - Then, we comment out the code in CUAppDelegate since ContactDetailViewController is no longer “root” and that code would now crash at run-time
- If you run the application, you’ll be presented with a completely empty table view; so, we need to add data

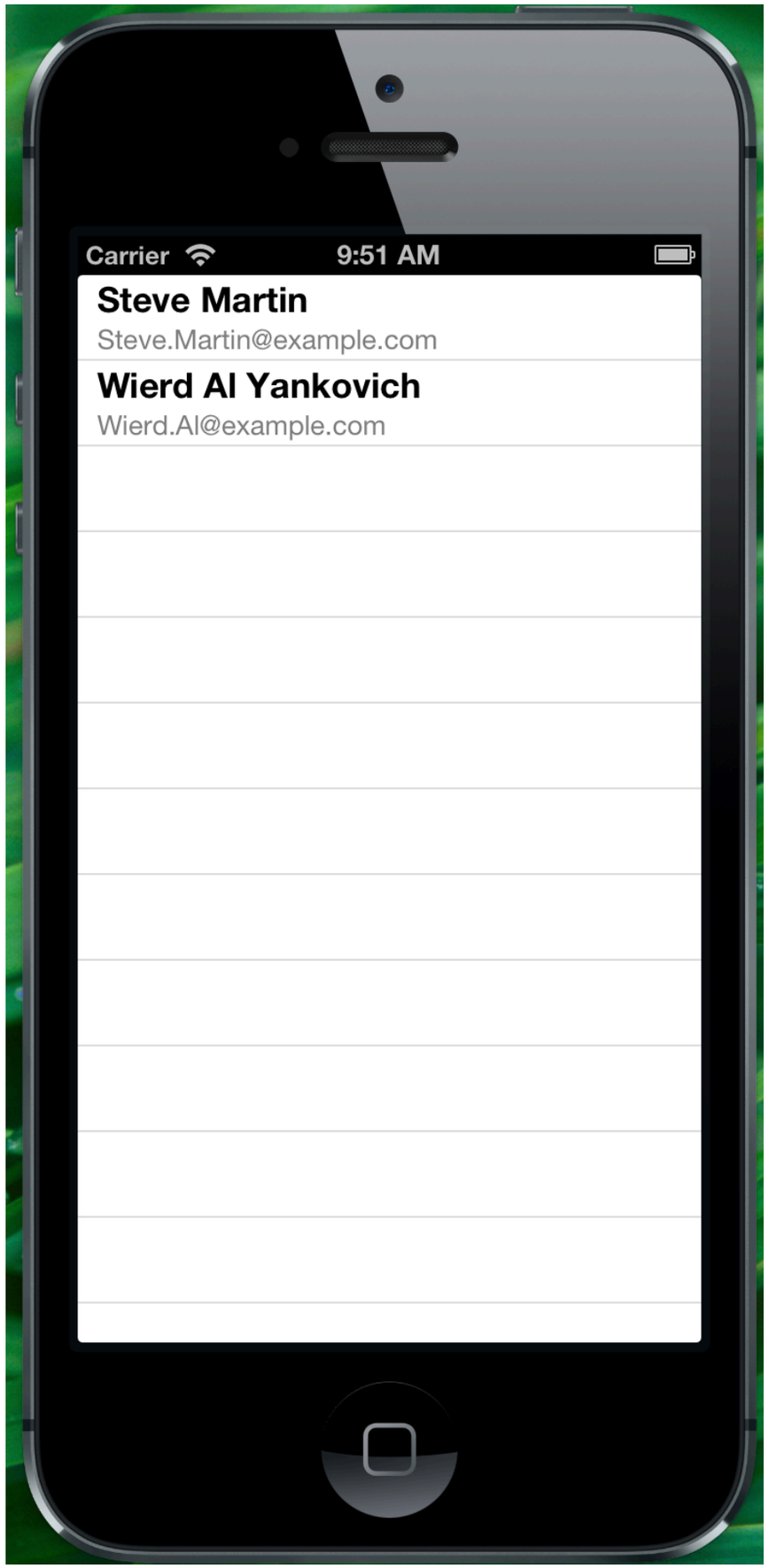
Steps to add data

- We will first make sure that our table view has some data to display
  - We will add a NSMutableArray property called contacts and initialize this array with two entries in the table view controller's viewDidLoad method
    - See example code for details
- Now, to get these contacts to display in the table view we need to implement the following methods
  - numberOfSectionsInTableView
  - numberOfRowsInSection
  - cellForRowAtIndexPath
- These methods are part of the "data source" protocol and are called to find out how much data to display in the table

```objc
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    return [self.contacts count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)
    indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier
        forIndexPath:indexPath];

    ContactInfo *info = [self.contacts objectAtIndex:indexPath.row];

    cell.textLabel.text = info.name;
    cell.detailTextLabel.text = info.email;

    return cell;
}
```

This is the relevant code; We have only 1 section. In that section, the number of rows is equal to the number of contacts.

The most complicated method is cellForRowAtIndexPath; We ask the table view for an instance of our “Cell” prototype. We then use the “row” property of the indexPath to retrieve a ContactInfo from contacts. We then configure the cell with information from that particular contact. This method is called once per row of the table.



Not bad for a few lines of code!

Here we can see that the table view has created two rows and its display the name and e-mail address of our two contacts

Let's connect this table view to the ContactDetailViewController

We need to create a modal segue for now with an identifier of "Show Details"; we'll switch to a push segue once we add our navigation controller

We will implement prepareForSegue to pass the selected row's contact information to the ContactDetailViewController

```objc
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([segue.identifier isEqualToString:@"Show Details"]) {
        ContactDetailViewController *vc = (ContactDetailViewController *)
            segue.destinationViewController;

        NSIndexPath *index = [self.tableView indexPathForCell:sender];

        ContactInfo *info = [self.contacts objectAtIndex:index.row];

        vc.info = info;
    }
}
```

After configuring the segue in Interface Builder, the prepareForSegue code looks like this

We make sure that the segue is the “Show Details” segue

We retrieve the destination view controller and cast it to the type we need

We ask the sender (our table cell) for its indexPath. We then use its “row” property to retrieve our contact and assign it to the retrieved view controller

Current Status

- If you run the application, you can now click on a table row and see the ContactDetailViewController slide into place
    - In addition, the ContactDetailViewController is filled with the appropriate information because we set its info property with the correct contact
- But, we have no overall navigation scheme in place
    - There is no way to return to the table view in the current set up
- So, now we are going to add a Navigation Controller to handle moving back and forth between the table view and the detail view

Embed in Navigation Controller

- Select the Table View in the storyboard and invoke the menu command
  - Editor ⇒ Embed In ⇒ Navigation Controller
- Interface Builder does the following
  - It creates a navigation controller
  - Makes it the root view controller
  - And sets the table view controllers as its initial view
- We can now go to our “Show Details” segue and convert it to a Push segue rather than a modal segue
  - Run the app and enjoy effortless navigation between our table view and detail view!

Polishing the App (I)

- Right now, our views when displayed in the navigation controller have no titles
    - If we set our two view controllers to have titles the app will look more polished
- Setting the title for the table view is easy
    - Select it in the storyboard, double click its top navigation bar and type in the title “Contacts”; You can run the app to see this change take effect
- To set the title for our detail view, we have to add this line of code in the viewWillAppear method of ContactDetailViewController
    - self.title = self.info.name;
- Run the app and check out both rows of the table in detail view

Polishing the App (II)

- The last thing we need to do is to save our edits such that they persist
  - Right now if you change anything in the detail view, those changes are discarded
    - The reason? We copied values from the model object into the widgets but we never copy those values back
    - To do that, we will add a viewDidDisappear: method to our ContactDetailViewController and copy all of the values out of the widgets and into the model object
    - If you run the app and change a contact's name and e-mail address, you will see that 1) the changes do not show up in the table but 2) the changes are made; if you reselect that contact, you'll see your changes in the detail view

Polishing the App (III)

- So, now, all that we need to do to “complete” the app is to notify the table view that the contact info has changed and so it needs to reload the table rows
- To do that, I’m going to add a property to ContactDetailViewController that will store a reference back to the Table View controller; we’ll set this property in prepareForSegue:
  - We’ll make that reference “weak” since we don’t own the table view controller
  - In viewDidDisappear we will use this reference to call a method that will reload the table
  - Any changes to name and/or e-mail address will now be displayed
    - See sample code for details

Are we done?

- No
  - We need to add the ability to create new contacts
  - But we don't have time for that in this lecture!

Wrapping Up

- Today we covered the following topics
    - Fundamentals of storyboards and view controllers
    - Container Controllers
        - Tab Bar controllers and Navigation Controllers
            - And how to use them inside of storyboards
- Table View Controllers
    - How to display data in them via model objects
    - How to pass a model object to a detailed view controller using segues

Coming Up Next

- Lecture 18: Intermediate Android
- Homework 4 will be released
  - It's the first step in the semester project
    - it will be due at the start of week 11