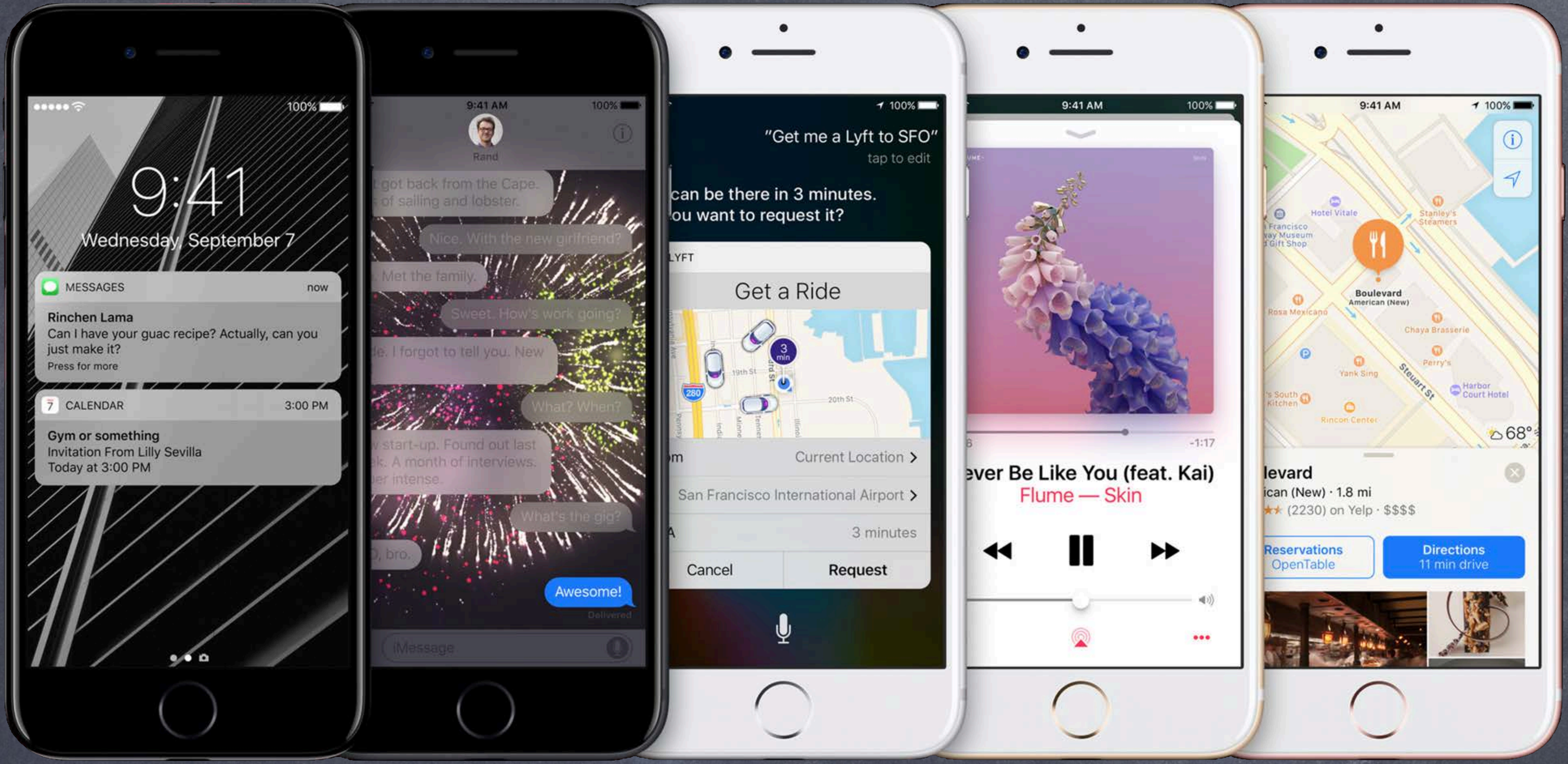


Stanford CS193p

Developing Applications for iOS
Winter 2017

Today

- Demo: The `FaceViewController` MVC's Model
  - It's a facial expression
- Gestures
  - Getting multitouch input from users
- Demo: Modifying the facial expression
  - Panning, pinching, tapping
- Multiple MVCs
  - Tab Bar, Navigation and Split View Controller

Demo

- The `FaceViewController` MVC's Model
  - It's a facial expression

Gestures

- We've seen how to draw in a `UIView`, how do we get touches?
  - We can get notified of the raw touch events (touch down, moved, up, etc.)
  - Or we can react to certain, predefined "gestures." The latter is the way to go!
- Gestures are recognized by instances of `UIGestureRecognizer`
  - The base class is "abstract." We only actually use concrete subclasses to recognize.
- There are two sides to using a gesture recognizer
  1. Adding a gesture recognizer to a `UIView` (asking the `UIView` to "recognize" that gesture)
  2. Providing a method to "handle" that gesture (not necessarily handled by the `UIView`)
- Usually the first is done by a Controller
  - Though occasionally a `UIView` will do this itself if the gesture is integral to its existence
- The second is provided either by the `UIView` or a `Controller`
  - Depending on the situation. We'll see an example of both in our demo.

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.

  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView ...

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's didSet code gets called when iOS hooks up this outlet at runtime

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's didSet code gets called when iOS hooks up this outlet at runtime
  Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's didSet code gets called when iOS hooks up this outlet at runtime
  Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
  The target gets notified when the gesture is recognized (here it's the Controller itself)

Gestures

- Adding a gesture recognizer to a UIView

  Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
  We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

  The property observer's didSet code gets called when iOS hooks up this outlet at runtime
  Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
  The target gets notified when the gesture is recognized (here it's the Controller itself)
  The action is the method invoked on recognition (that method's argument? the recognizer)

Gestures

Adding a gesture recognizer to a UIView

Imagine we wanted a UIView in our Controller's View to recognize a "pan" gesture.
We can configure it to do so in the property observer for the outlet to that UIView …

```
@IBOutlet weak var pannableView: UIView {
    didSet {
        let panGestureRecognizer = UIPanGestureRecognizer(
            target: self, action: #selector(ViewController.pan(recognizer:))
        )
        pannableView.addGestureRecognizer(panGestureRecognizer)
    }
}
```

The property observer's didSet code gets called when iOS hooks up this outlet at runtime
Here we are creating an instance of a concrete subclass of UIGestureRecognizer (for pans)
The target gets notified when the gesture is recognized (here it's the Controller itself)
The action is the method invoked on recognition (that method's argument? the recognizer)
Here we ask the UIView to actually start trying to recognize this gesture in its bounds
Let's talk about how we implement the handler …

Gestures

- A handler for a gesture needs gesture-specific information

  So each concrete subclass provides special methods for handling that type of gesture

- For example, `UIPanGestureRecognizer` provides 3 methods

  ```
  func translation(in: UIView?) -> CGPoint // cumulative since start of recognition
  func velocity(in: UIView?) -> CGPoint     // how fast the finger is moving (points/s)
  func setTranslation(CGPoint, in: UIView?)
  ```

  This last one is interesting because it allows you to reset the translation so far

  By resetting the translation to zero all the time, you end up getting "incremental" translation

- The abstract superclass also provides state information

  ```
  var state: UIGestureRecognizerState { get }
  ```

  This sits around in `.possible` until recognition starts

  For a continuous gesture (e.g. pan), it moves from `.began` thru repeated `.changed` to `.ended`

  For a discrete (e.g. a swipe) gesture, it goes straight to `.ended` or `.recognized`.

  It can go to `.failed` or `.cancelled` too, so watch out for those!

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(recognizer:)`

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the action was pan(recognizer:)
We are only going to do anything when the finger moves or lifts up off the device's surface

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(recognizer:)`
We are only going to do anything when the finger moves or lifts up off the device's surface
`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(recognizer:)`
We are only going to do anything when the finger moves or lifts up off the device's surface
`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)
Here we get the location of the pan in the `pannableView`'s coordinate system

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(recognizer:)`
We are only going to do anything when the finger moves or lifts up off the device's surface
`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)
Here we get the location of the pan in the `pannableView`'s coordinate system
Now we do whatever we want with that information

Gestures

- So, given this information, what would the pan handler look like?

```
func pan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
        case .changed: fallthrough
        case .ended:
            let translation = recognizer.translation(in: pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            recognizer.setTranslation(CGPoint.zero, in: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(recognizer:)`
We are only going to do anything when the finger moves or lifts up off the device's surface
`fallthrough` is "execute the code for the next case down" (`case .changed,.ended:` ok too)
Here we get the location of the pan in the `pannableView`'s coordinate system
Now we do whatever we want with that information
By resetting the `translation`, the next one we get will be incremental movement

Gestures

- UIPinchGestureRecognizer

```
var scale: CGFloat                 // not read-only (can reset)
var velocity: CGFloat { get }      // scale factor per second
```

- UIRotationGestureRecognizer

```
var rotation: CGFloat              // not read-only (can reset); in radians
var velocity: CGFloat { get }      // radians per second
```

- UISwipeGestureRecognizer

  Set up the direction and number of fingers you want

```
var direction: UISwipeGestureRecoginzerDirection  // which swipe directions you want
var numberOfTouchesRequired: Int                  // finger count
```

- UITapGestureRecognizer

  Set up the number of taps and fingers you want

```
var numberOfTapsRequired: Int     // single tap, double tap, etc.
var numberOfTouchesRequired: Int // finger count
```

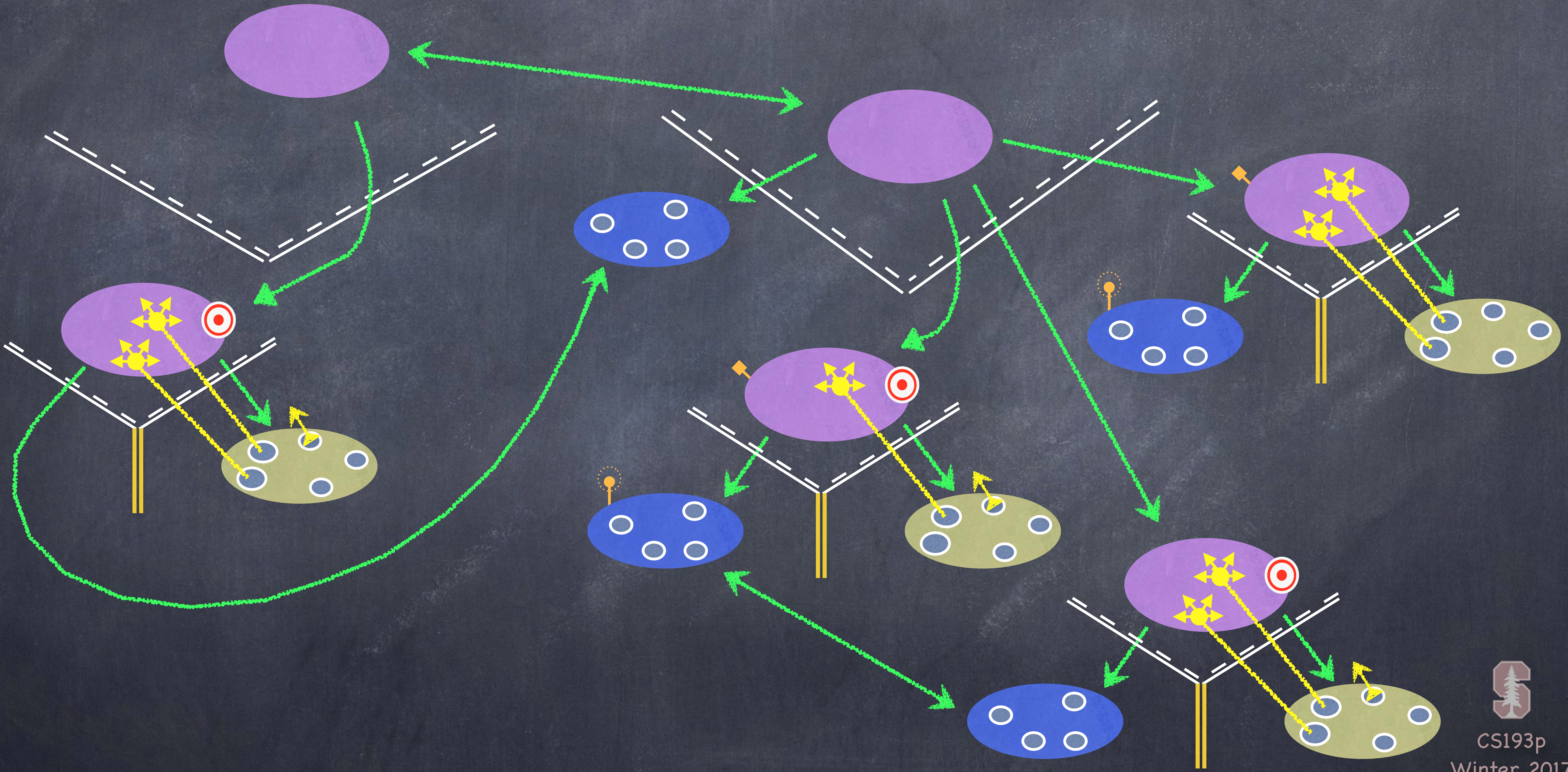
Demo

- Gestures Demo

  Add a gesture recognizer (pinch) to zoom in and out (control the FaceView's own scale)

  Add gesture recognizers (pan & tap) to control the expression (Model) in the Controller

MVCs working together

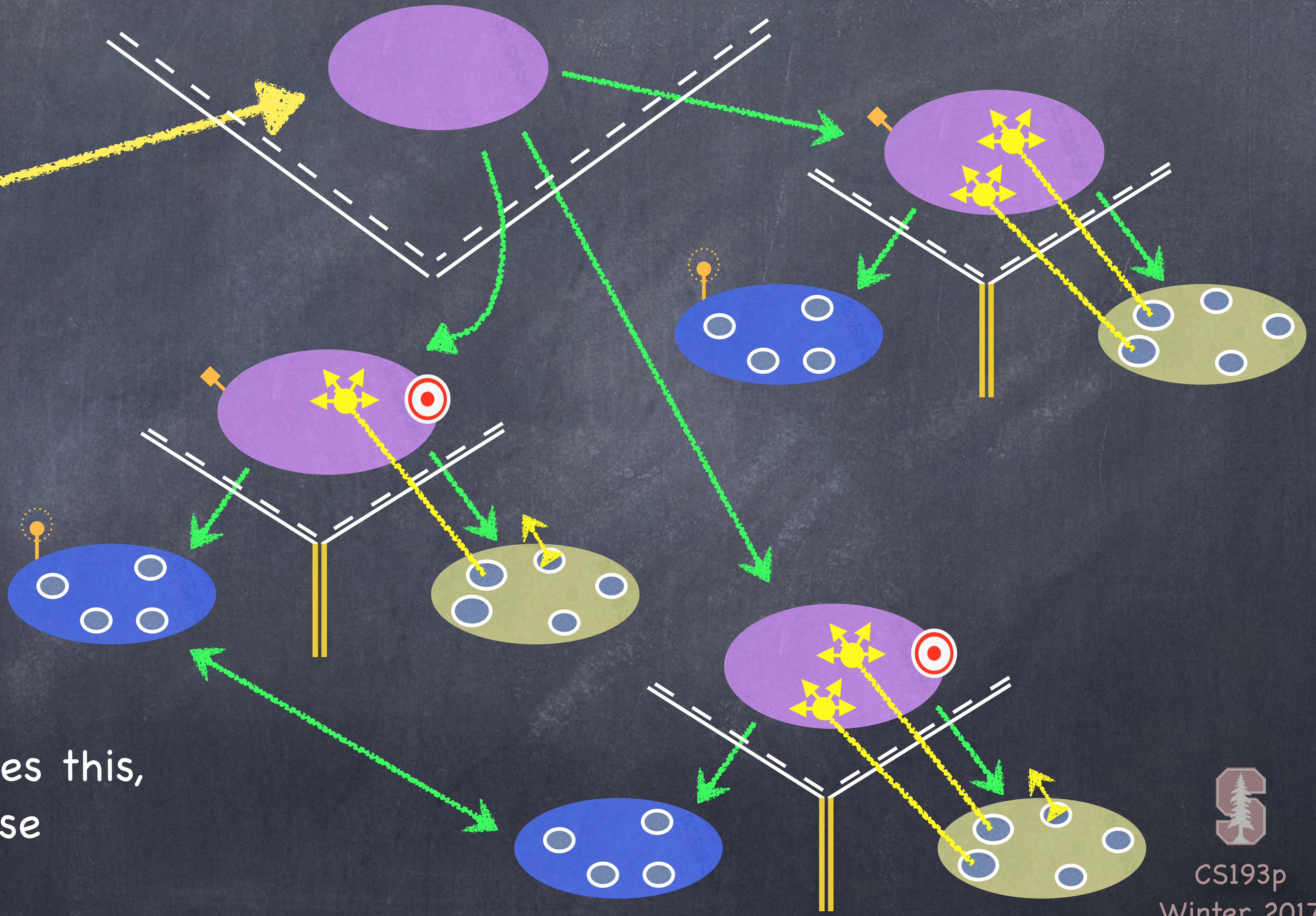


Multiple MVCs

- Time to build more powerful applications

  To do this, we must combine MVCs …

iOS provides some Controllers whose View is "other MVCs" *



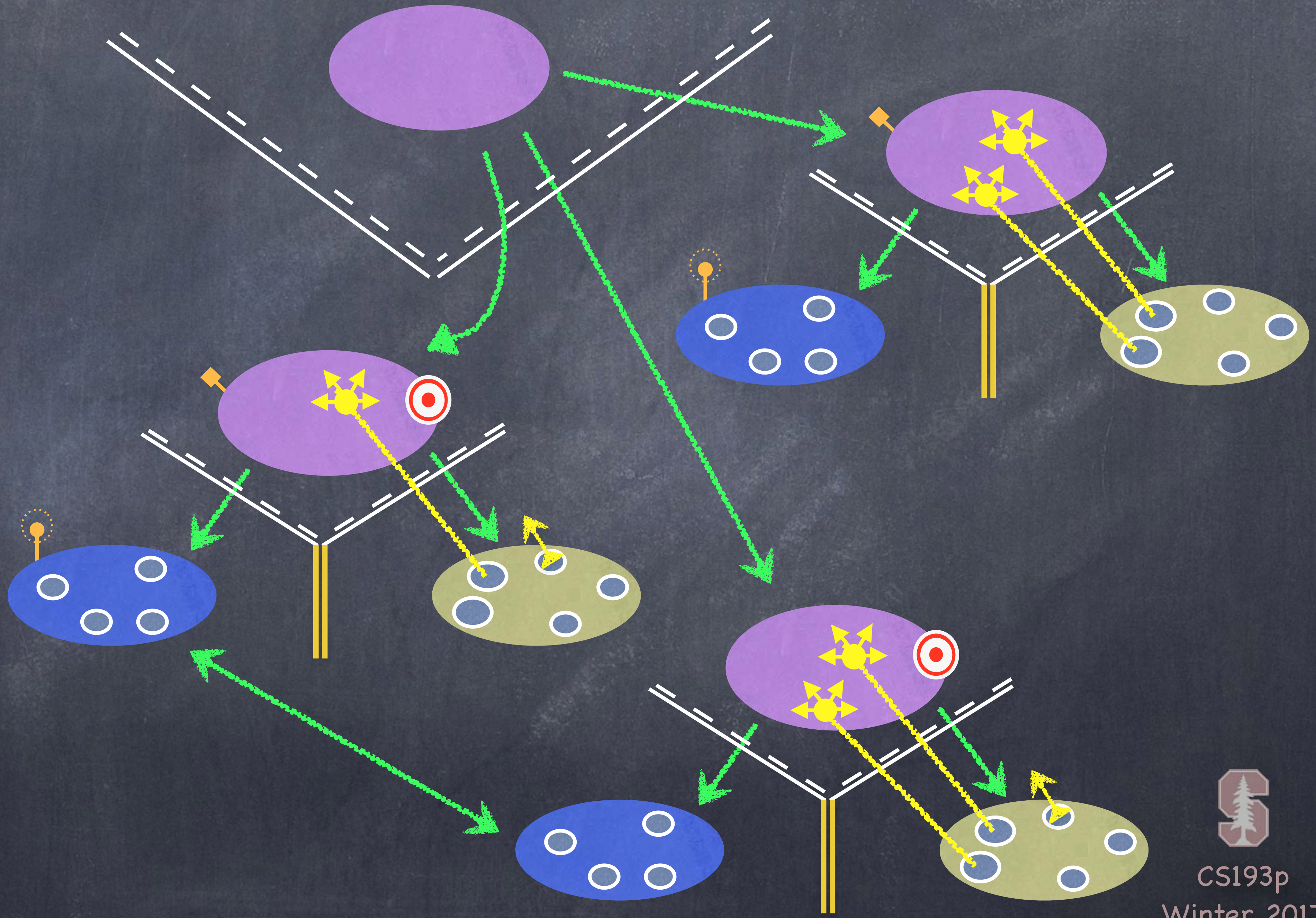
* you could build your own Controller that does this, but we're not going to cover that in this course

Multiple MVCs

- Time to build more powerful applications

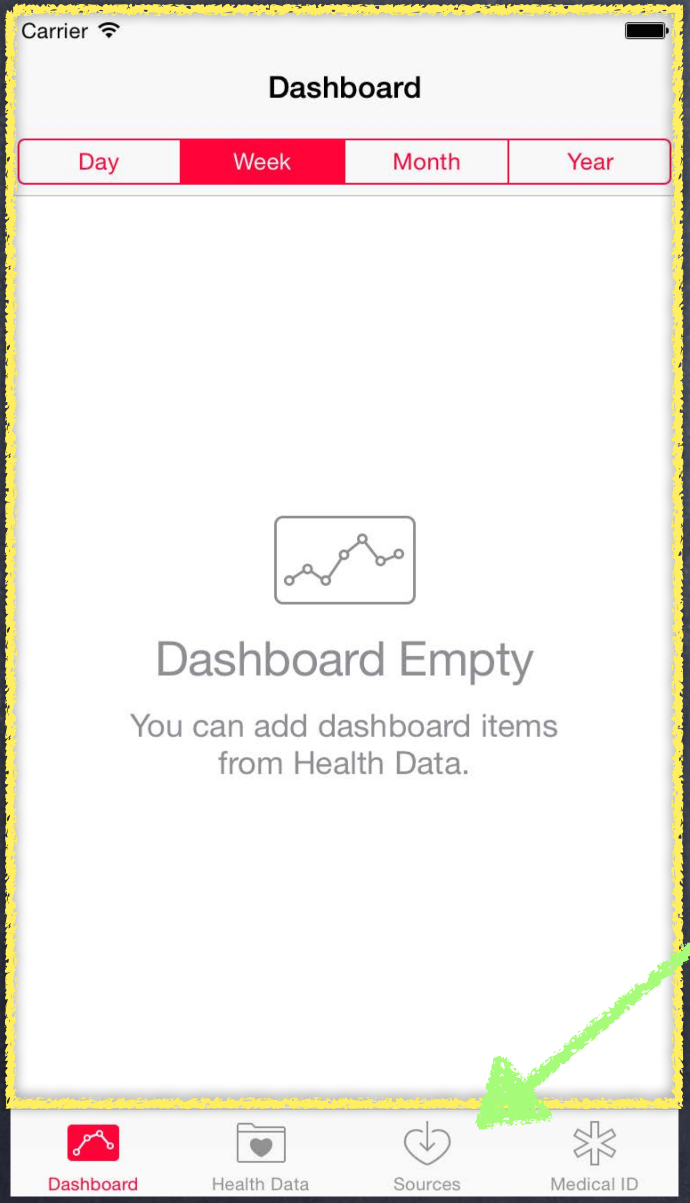
  To do this, we must combine MVCs …

  iOS provides some Controllers
  whose View is "other MVCs"
  Examples:
  UITabBarController
  UISplitViewController
  UINavigationController



UITabBarController

- It lets the user choose between different MVCs …



A "Dashboard" MVC

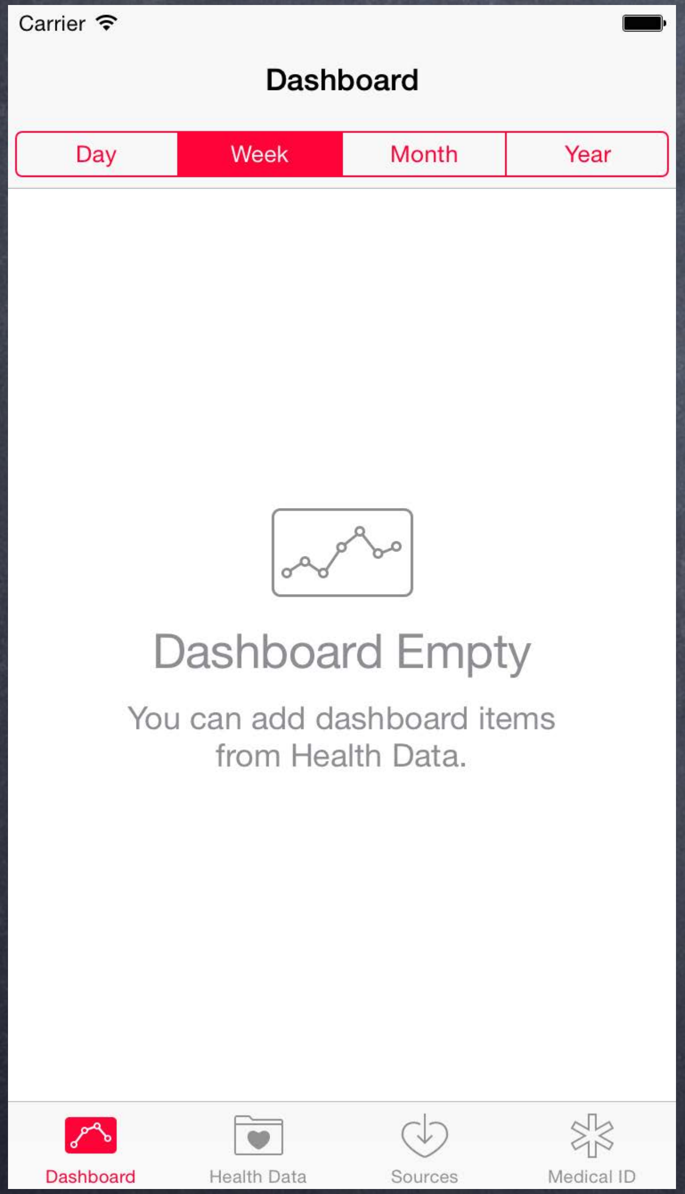
The icon, title and even a "badge value" on these is determined by the MVCs themselves via their property:

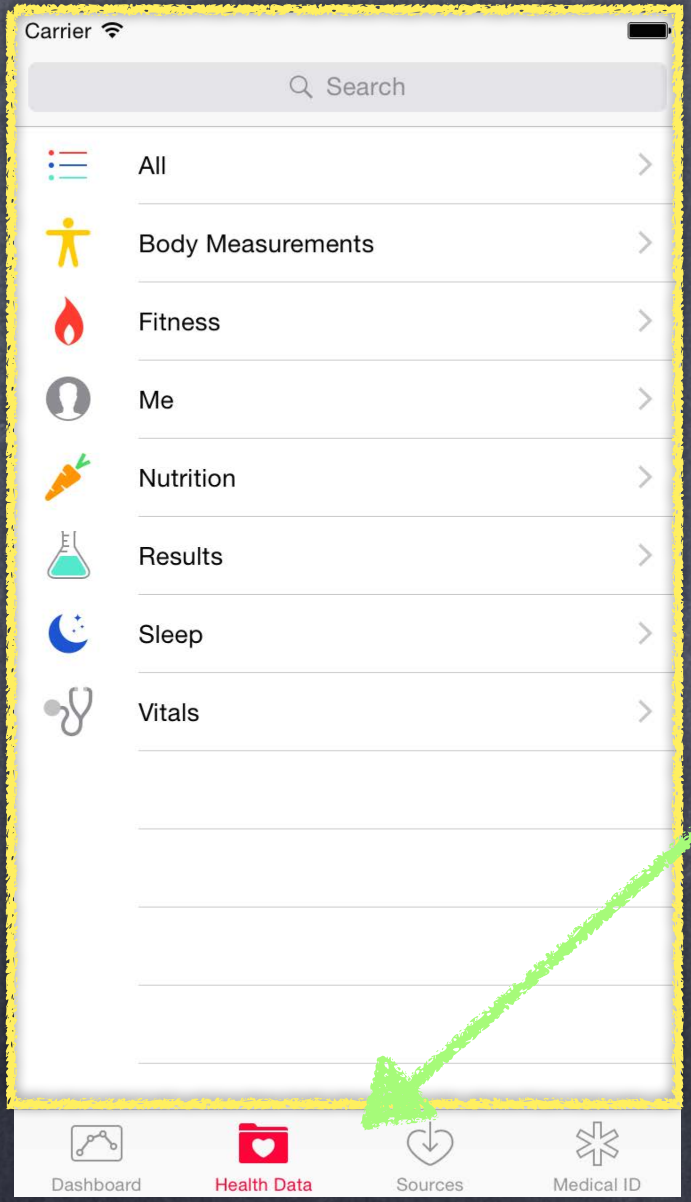
var tabBarItem: UITabBarItem!

But usually you just set them in your storyboard.

UITabBarController

- It lets the user choose between different MVCs …



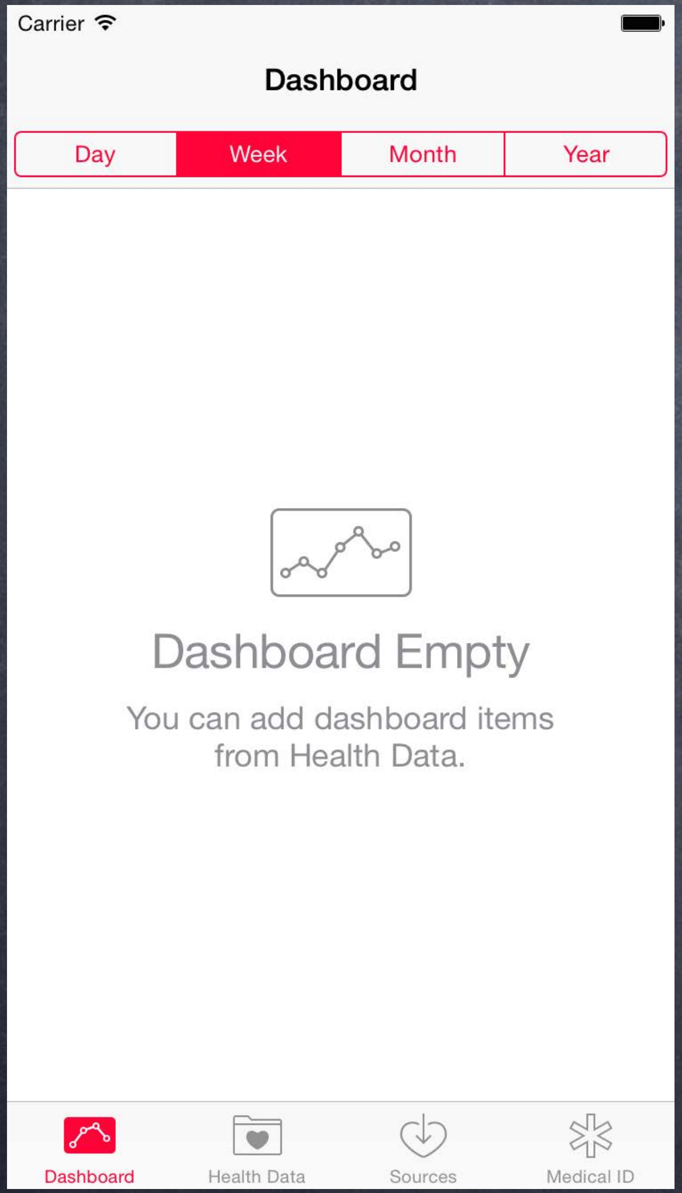


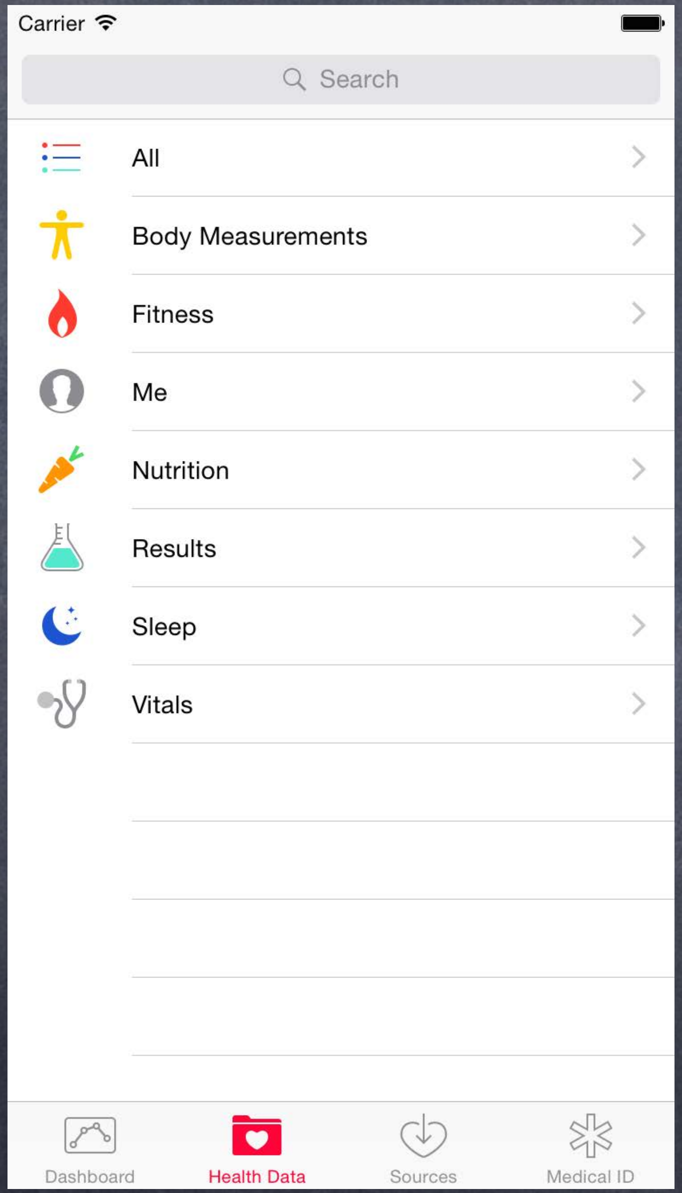
A "Health Data" MVC

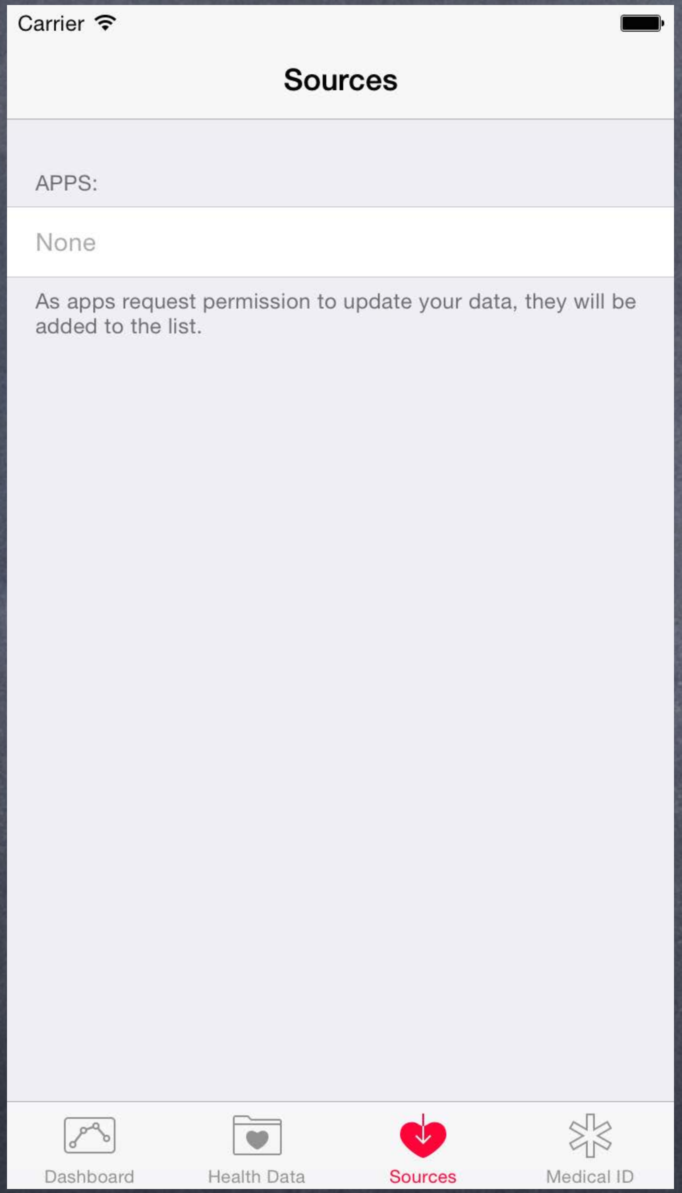
If there are too many tabs to fit here, the UITabBarController will automatically present a UI for the user to manage the overflow!

UITabBarController

- It lets the user choose between different MVCs …



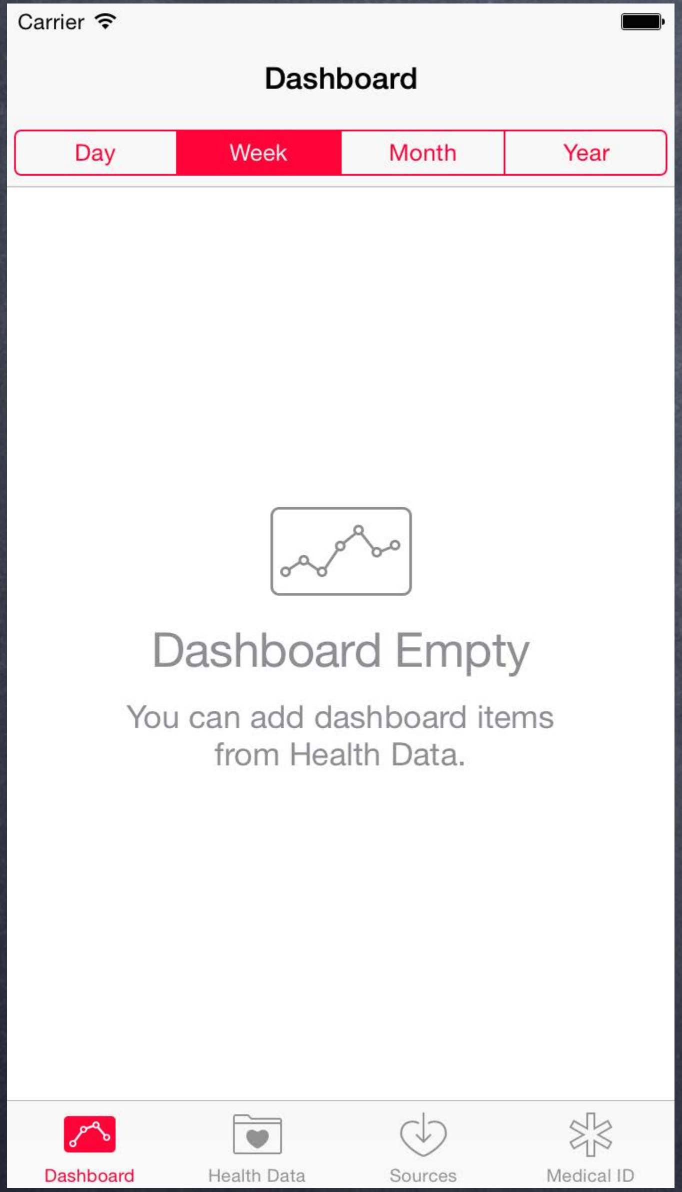




Carrier
Sources
APPS:
None
As apps request permission to update your data, they will be added to the list.
Dashboard
Health Data
Sources
Medical ID

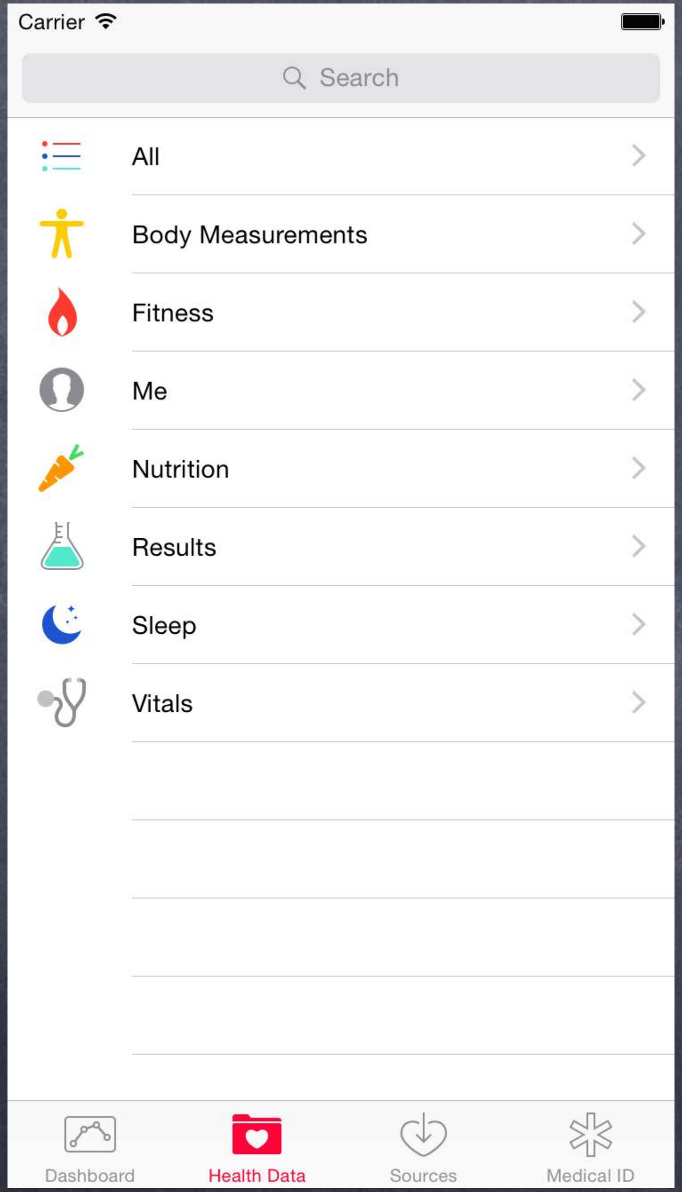
UITabBarController

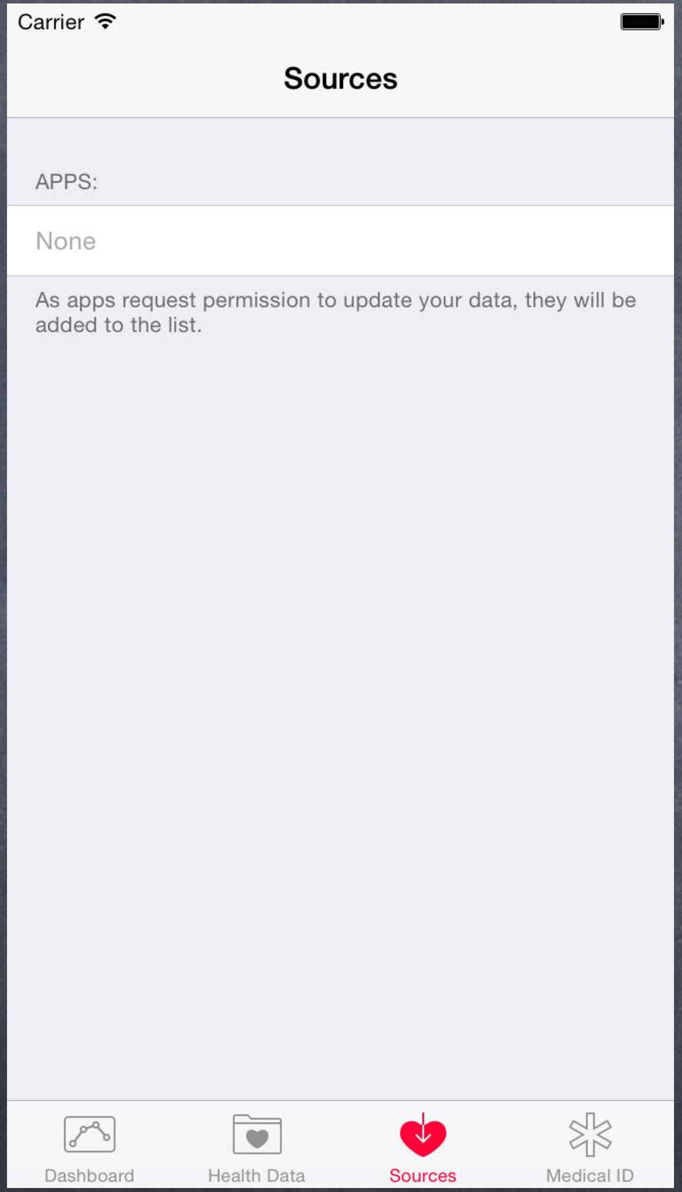
- It lets the user choose between different MVCs …



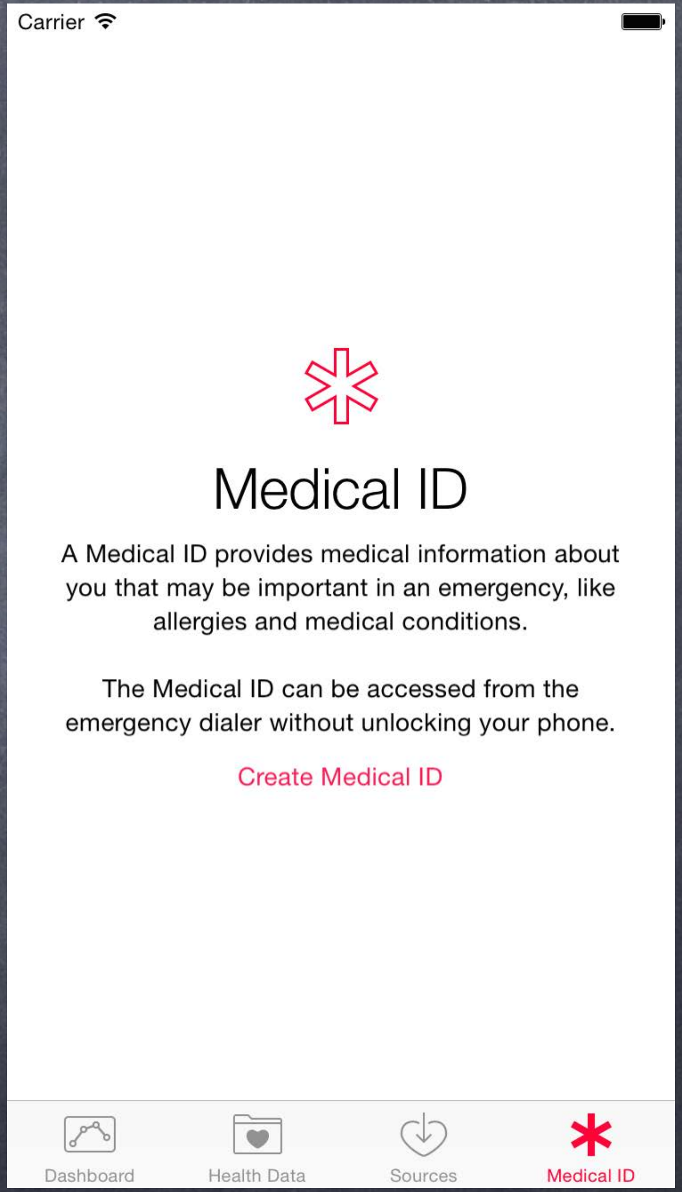
Dashboard Empty

You can add dashboard items from Health Data.





As apps request permission to update your data, they will be added to the list.



Medical ID

A Medical ID provides medical information about you that may be important in an emergency, like allergies and medical conditions.

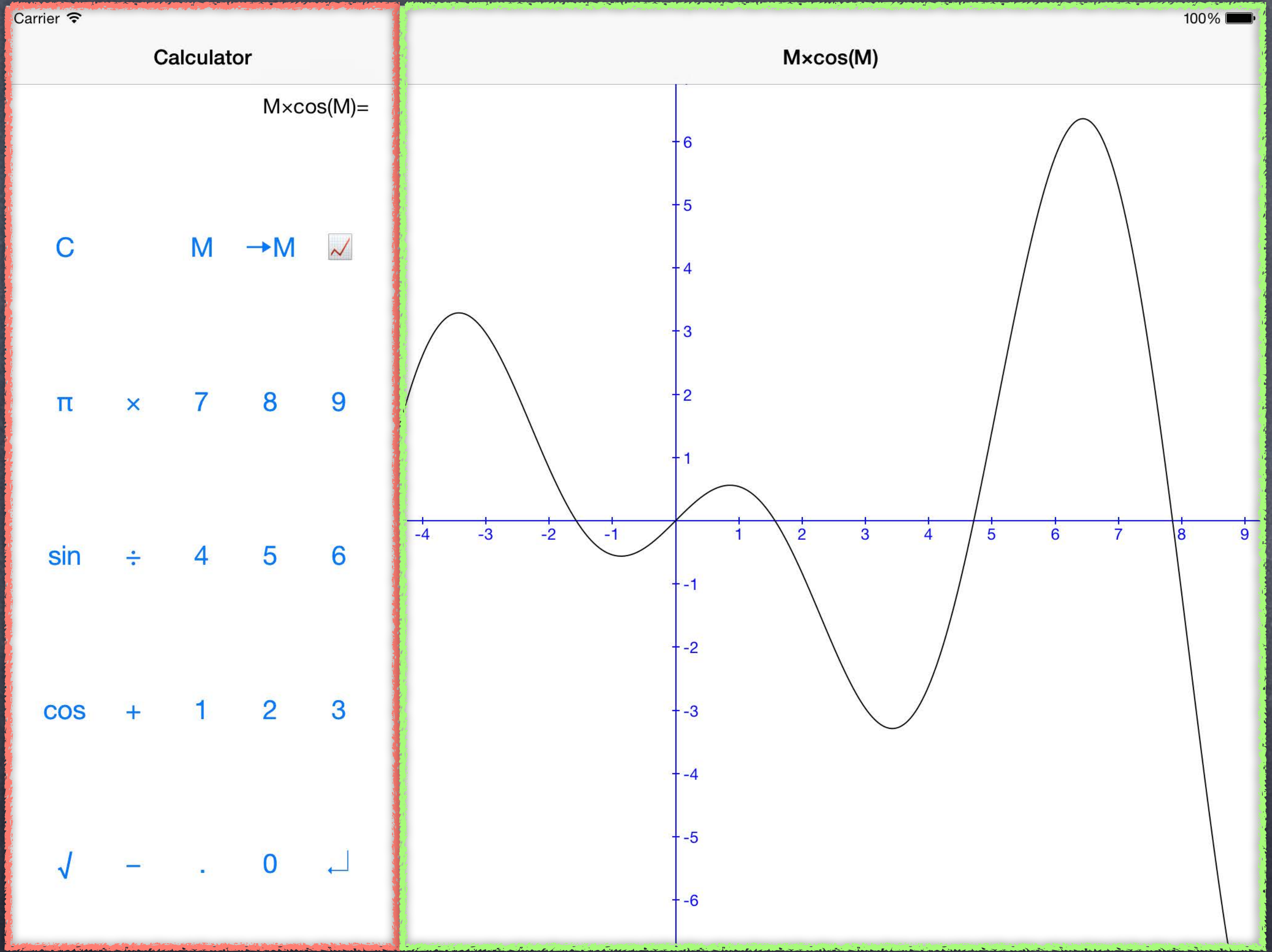
The Medical ID can be accessed from the emergency dialer without unlocking your phone.

UISplitViewController

- Puts two MVCs side-by-side …

A Calculator MVC

Master



A Calculator Graph MVC

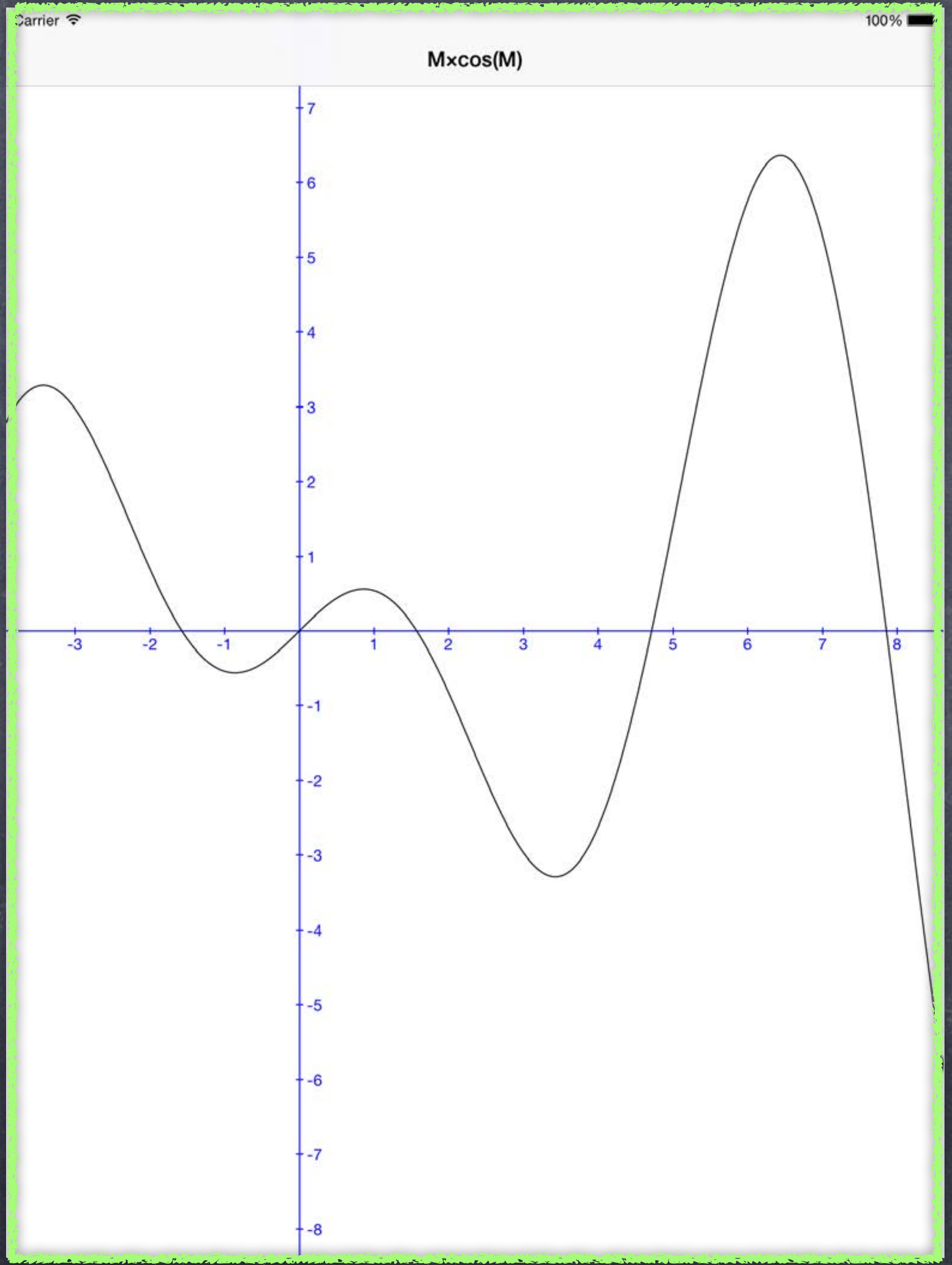
Detail

UISplitViewController

- Puts two MVCs side-by-side …

A
Calculator
MVC

Master



A
Calculator Graph
MVC

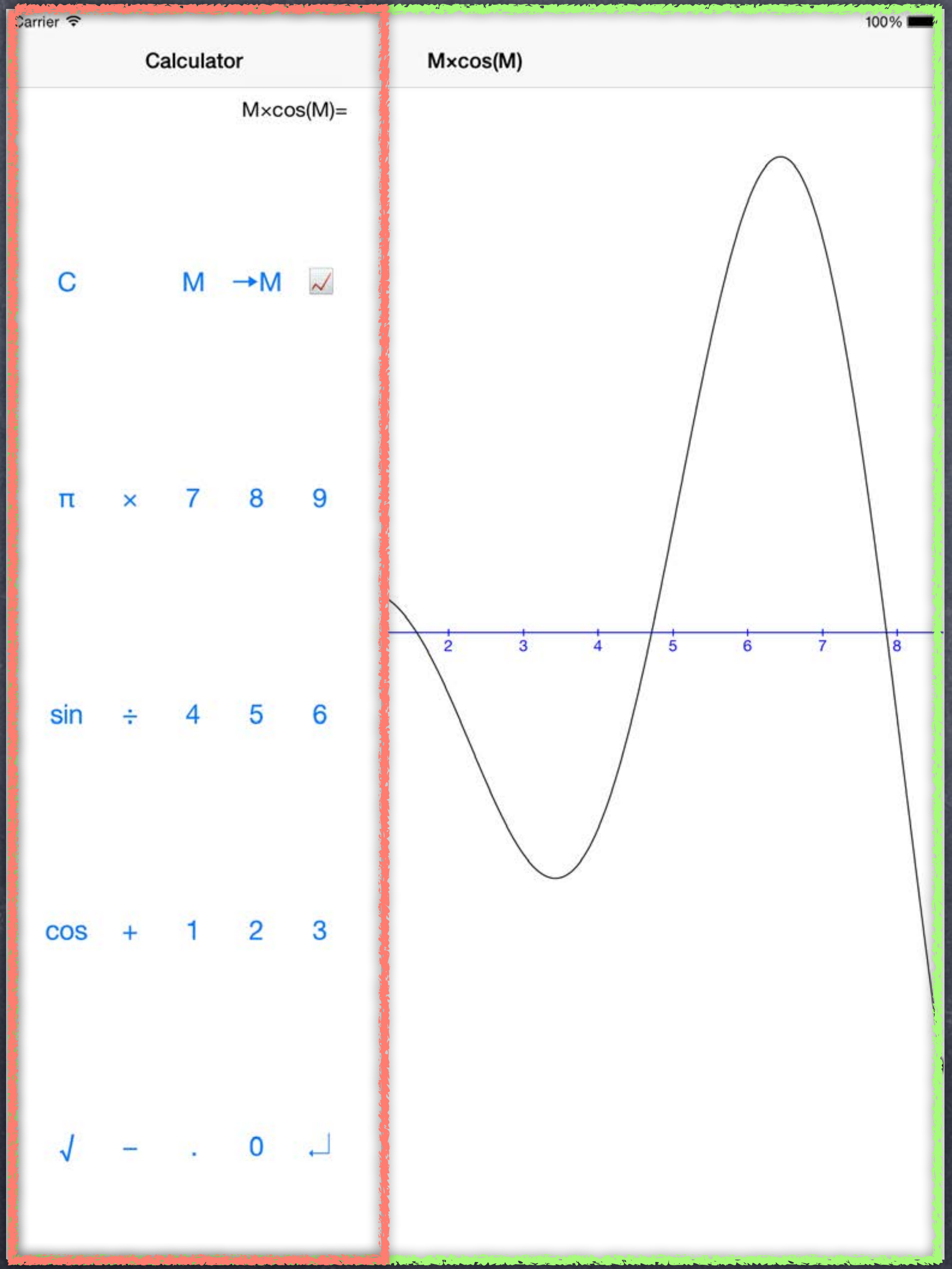
Detail

UISplitViewController

- Puts two MVCs side-by-side ...

A Calculator MVC

Master



A Calculator Graph MVC

Detail

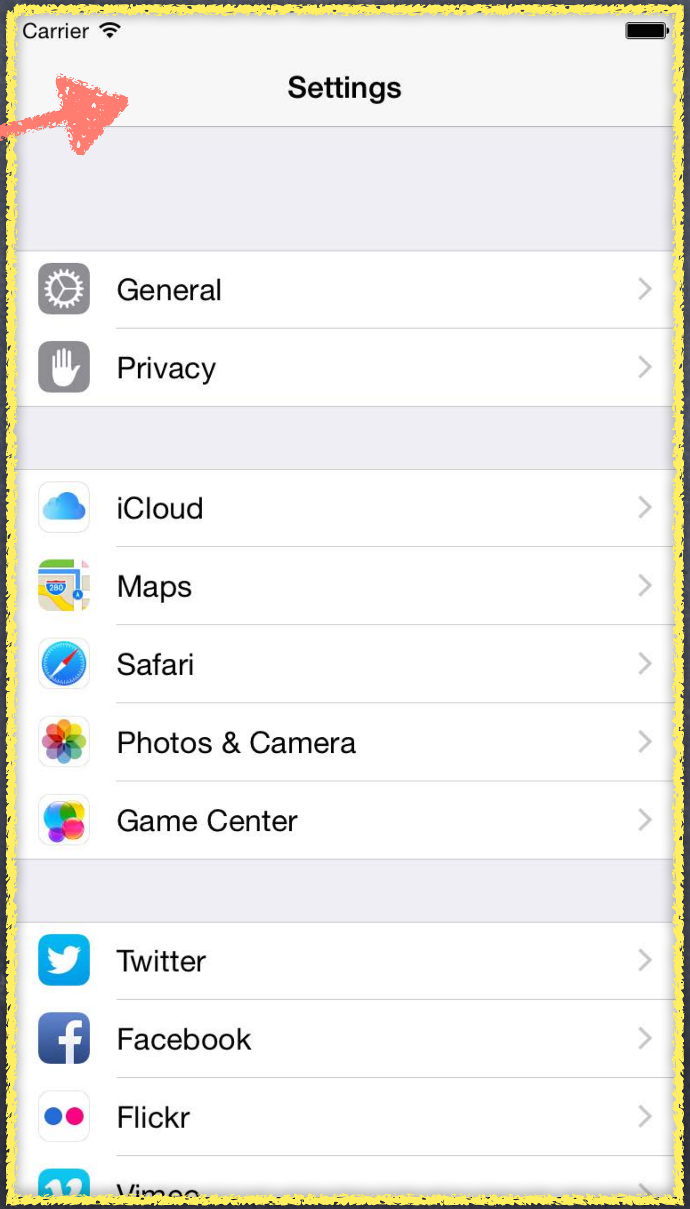
UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …

This top area is drawn by the UINavigationController

But the contents of the top area (like the title or any buttons on the right) are determined by the MVC currently showing (in this case, the "All Settings" MVC)

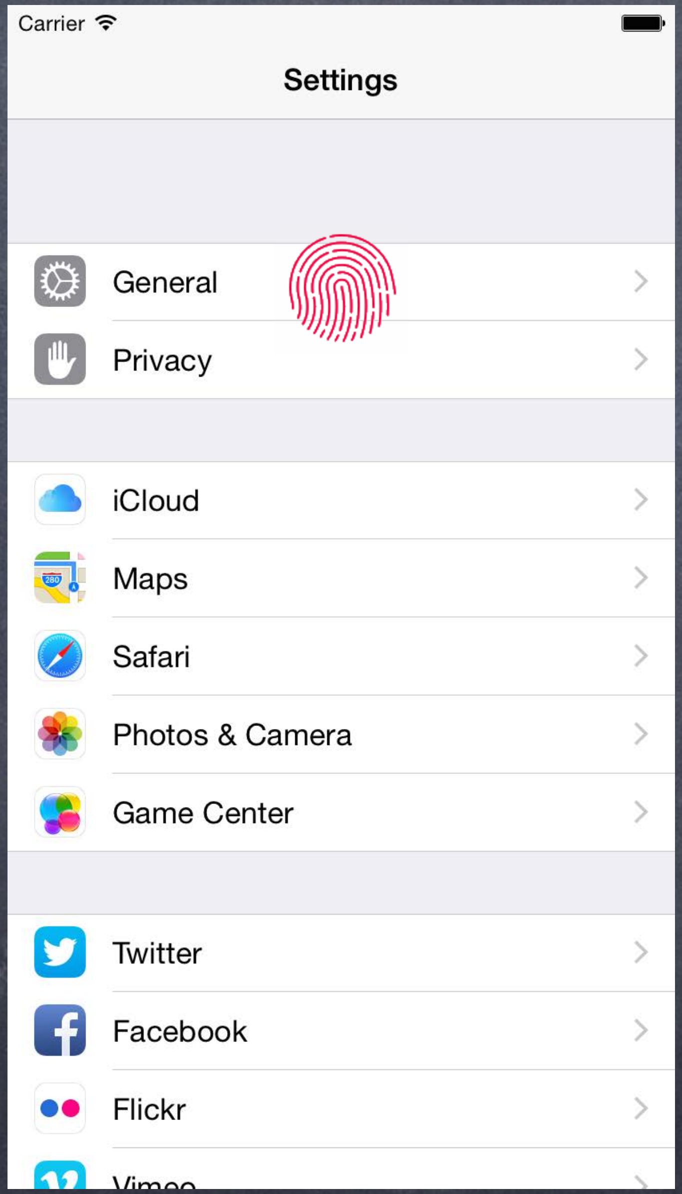
Each MVC communicates these contents via its UIViewController's navigationItem property



An "All Settings" MVC

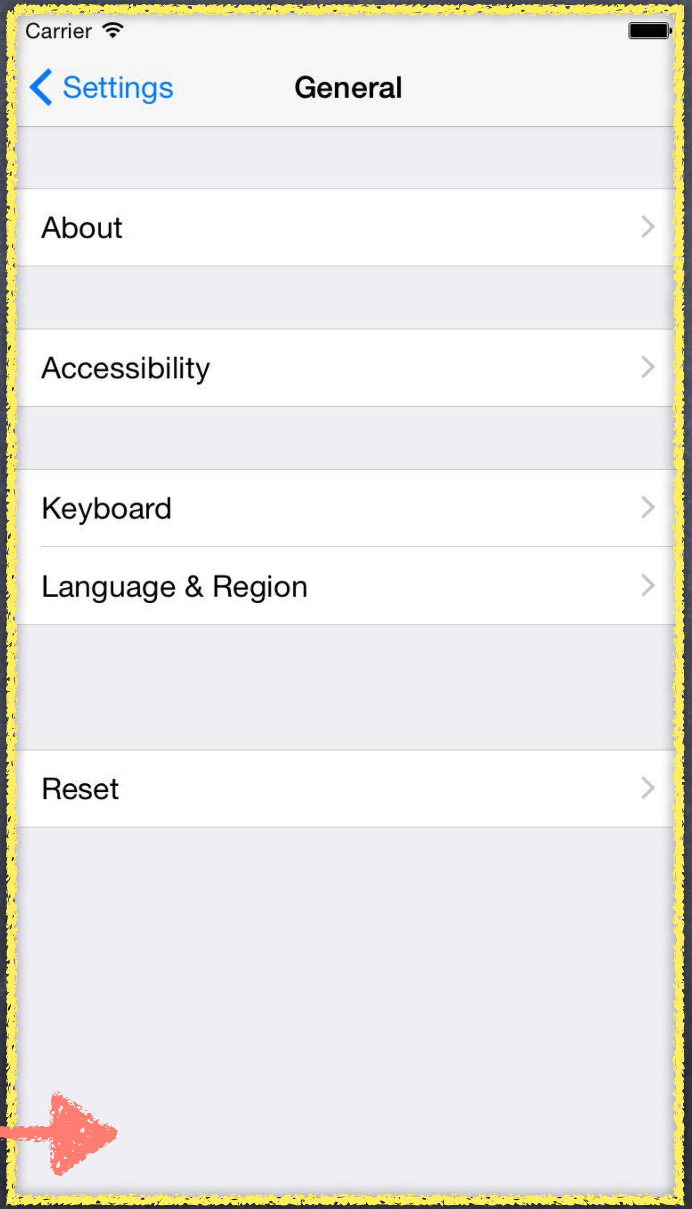
UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) ...



UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …



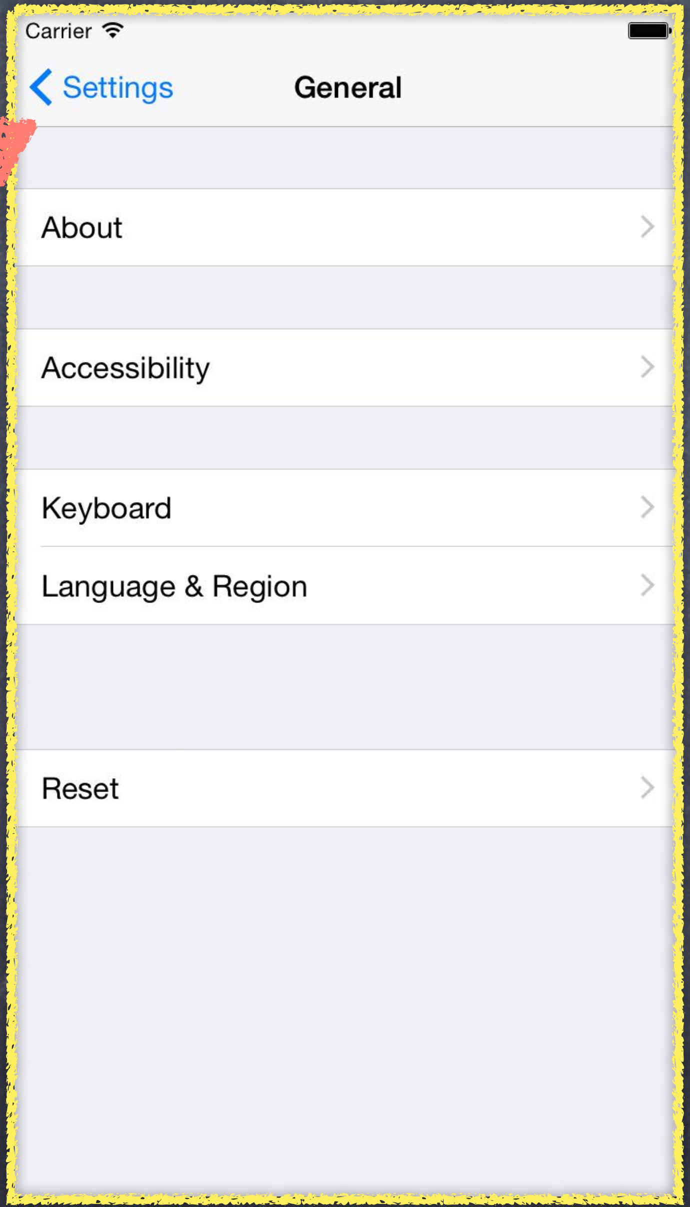
A "General Settings" MVC

It's possible to add MVC-specific buttons here too via the `UIViewController`'s `toolbarItems` property

UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …

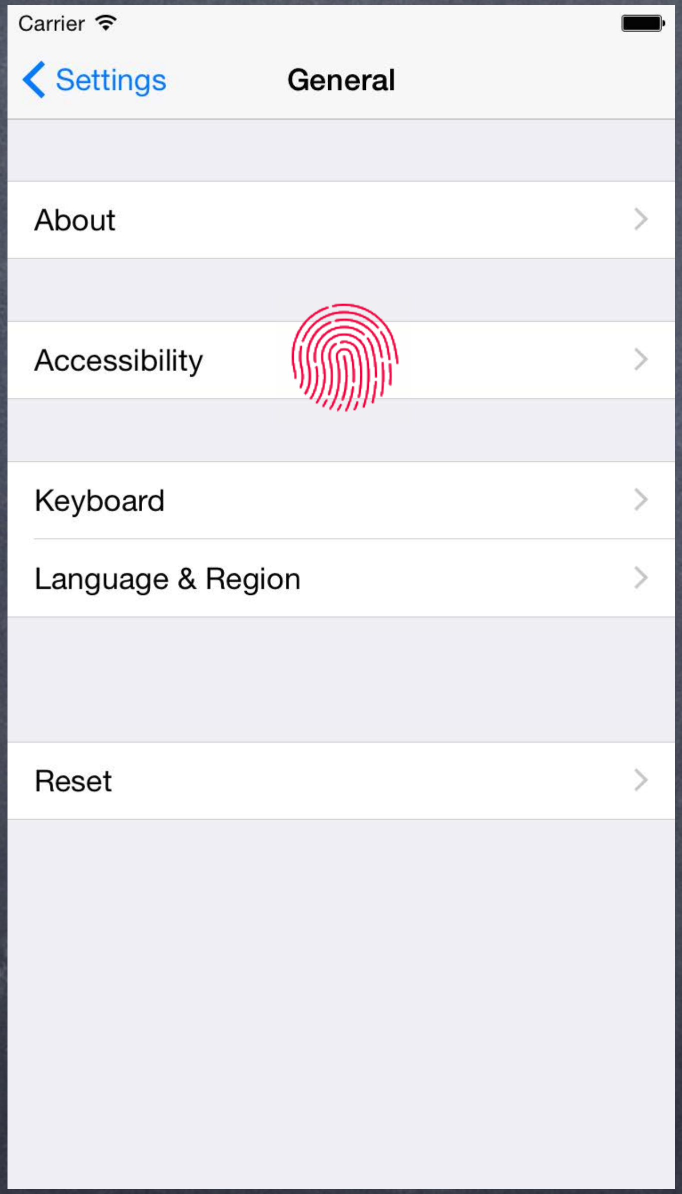
Notice this "back" button has appeared. This is placed here automatically by the UINavigationController.



A "General Settings" MVC

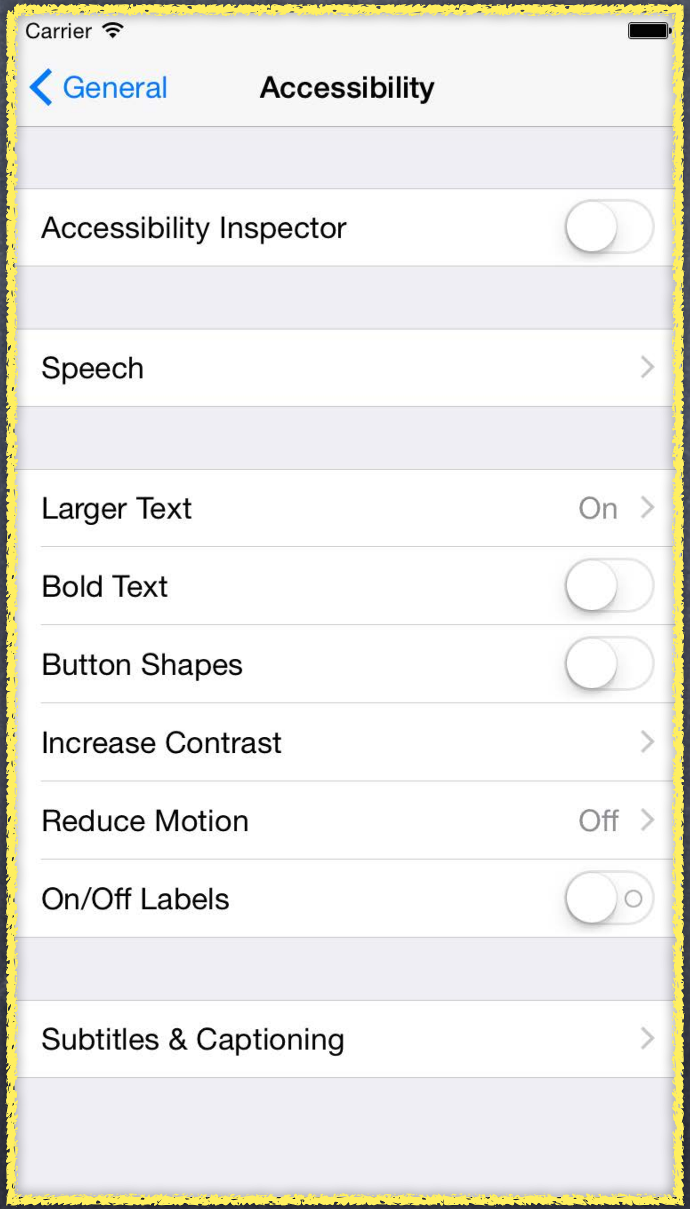
UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …



UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …



Carrier

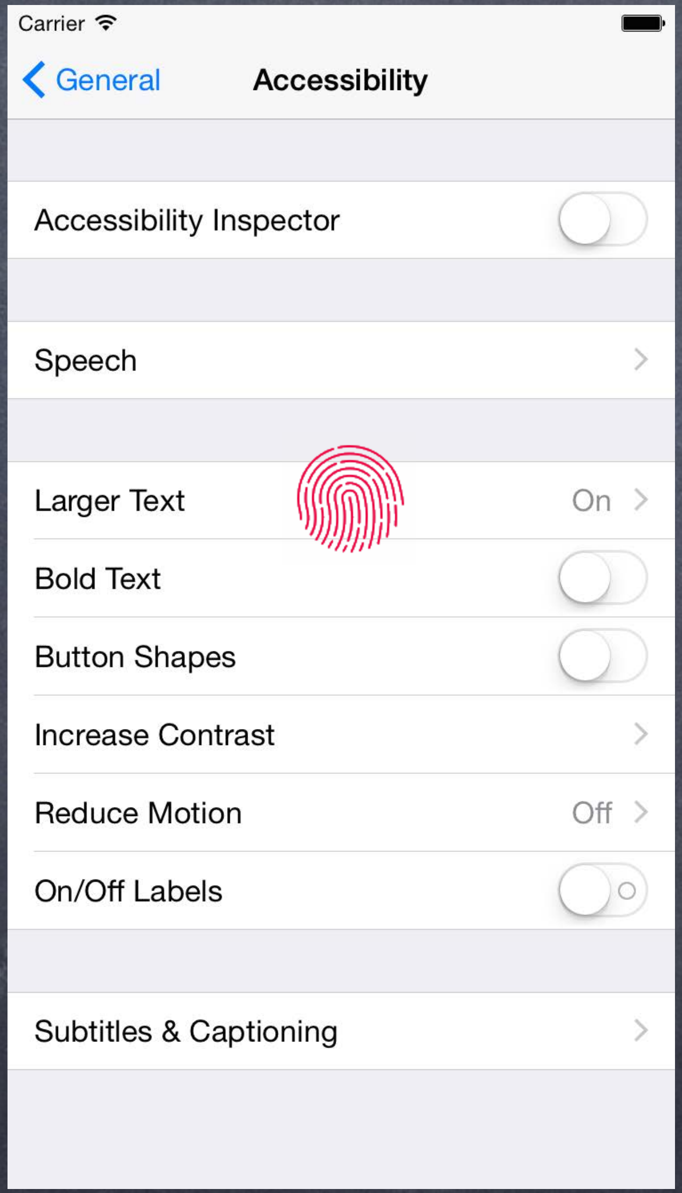
General Accessibility

Accessibility Inspector	
Speech	>
Larger Text	On >
Bold Text	
Button Shapes	
Increase Contrast	>
Reduce Motion	Off >
On/Off Labels	
Subtitles & Captioning	>

An "Accessibility" MVC

UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …



Accessibility Inspector

Speech

Larger Text On

Bold Text

Button Shapes

Increase Contrast

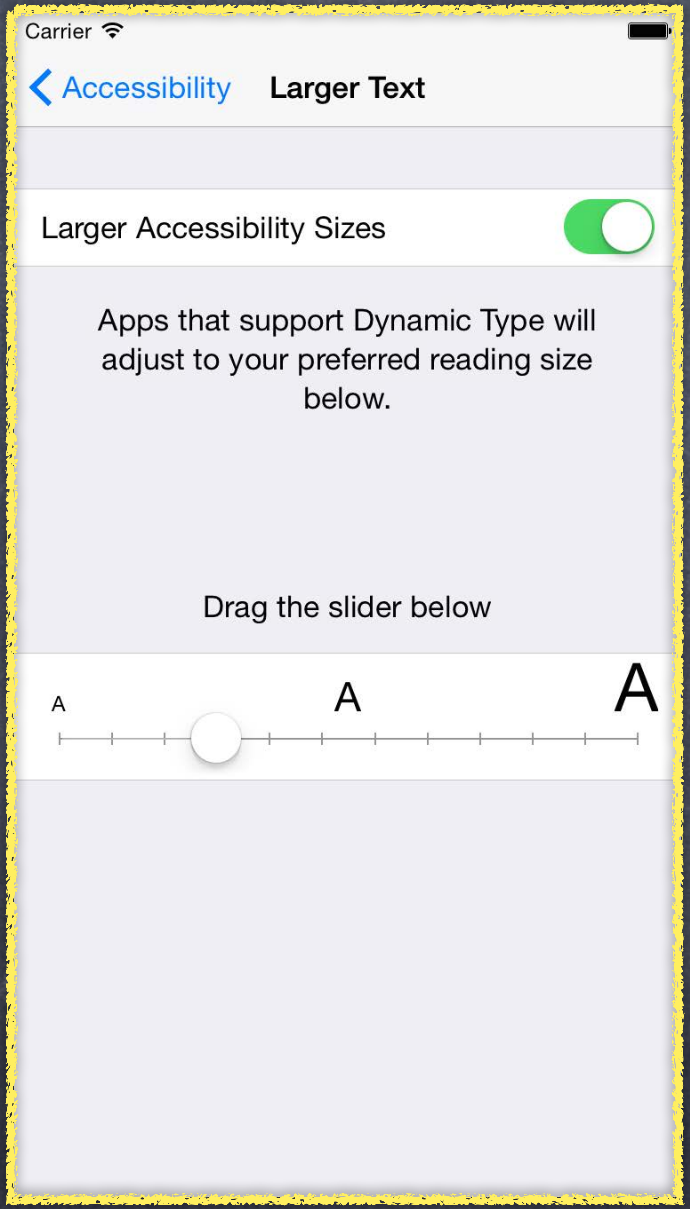
Reduce Motion Off

On/Off Labels

Subtitles & Captioning

UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …



Larger Accessibility Sizes

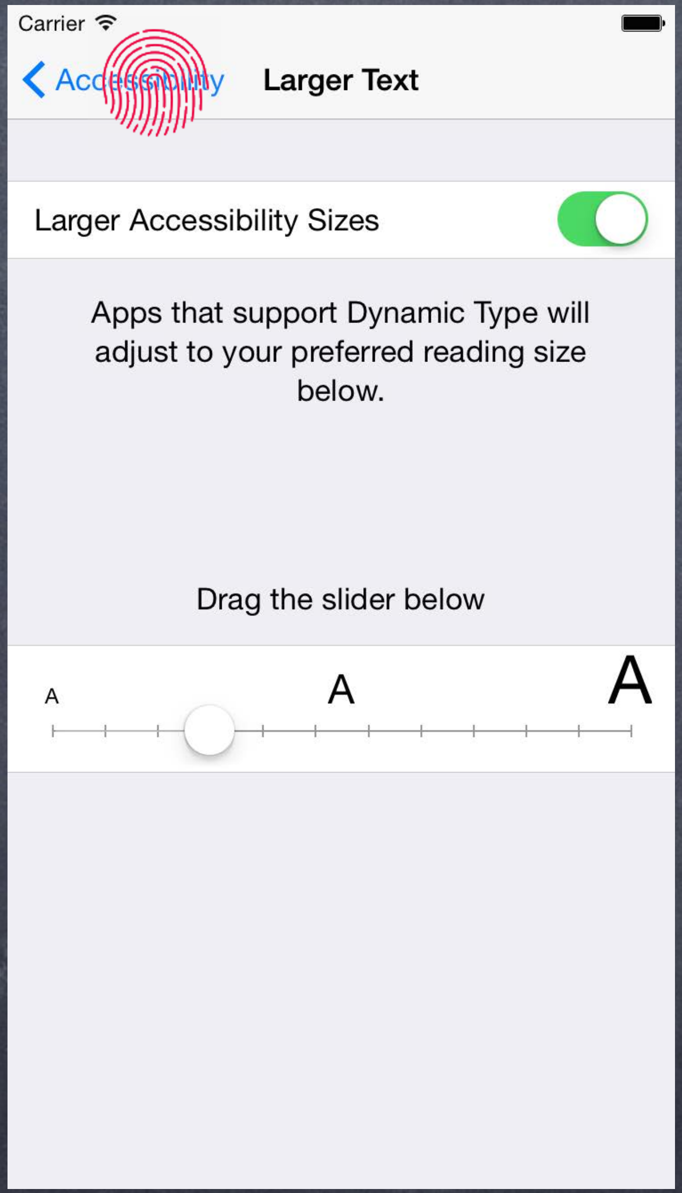
Apps that support Dynamic Type will adjust to your preferred reading size below.

Drag the slider below

A "Larger Text" MVC

UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …



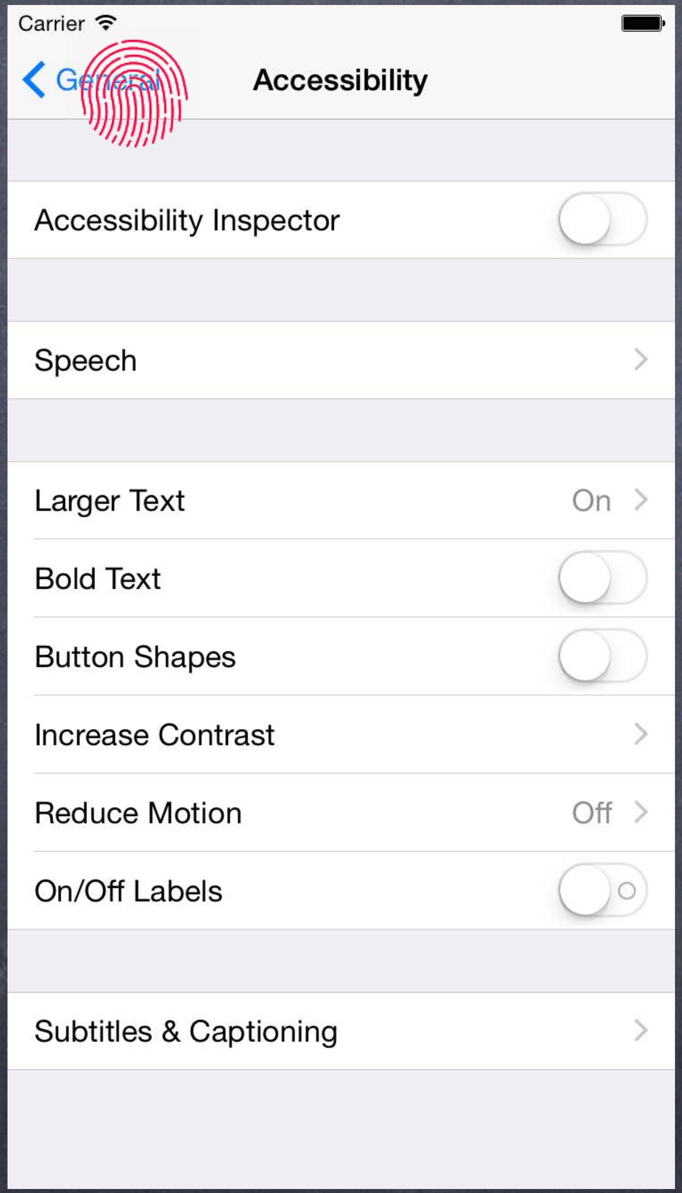
Larger Accessibility Sizes

Apps that support Dynamic Type will adjust to your preferred reading size below.

Drag the slider below

UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …

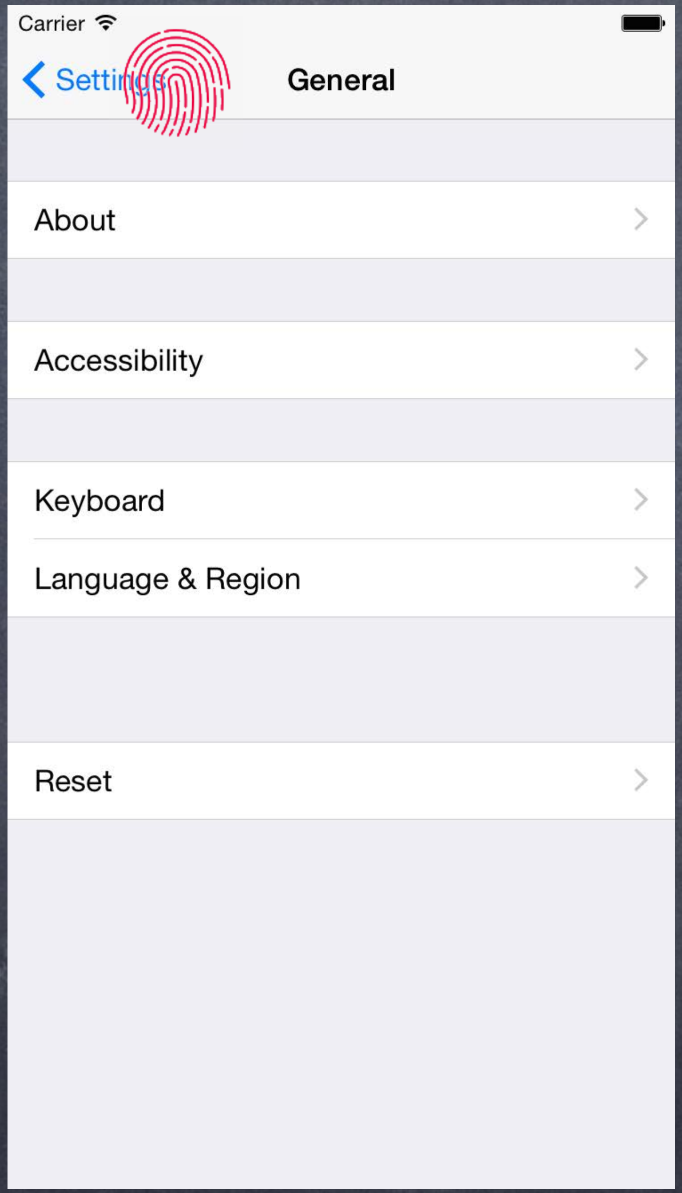

Carrier

General | Accessibility

Accessibility Inspector	
Speech	>
Larger Text	On >
Bold Text	
Button Shapes	
Increase Contrast	>
Reduce Motion	Off >
On/Off Labels	
Subtitles & Captioning	>

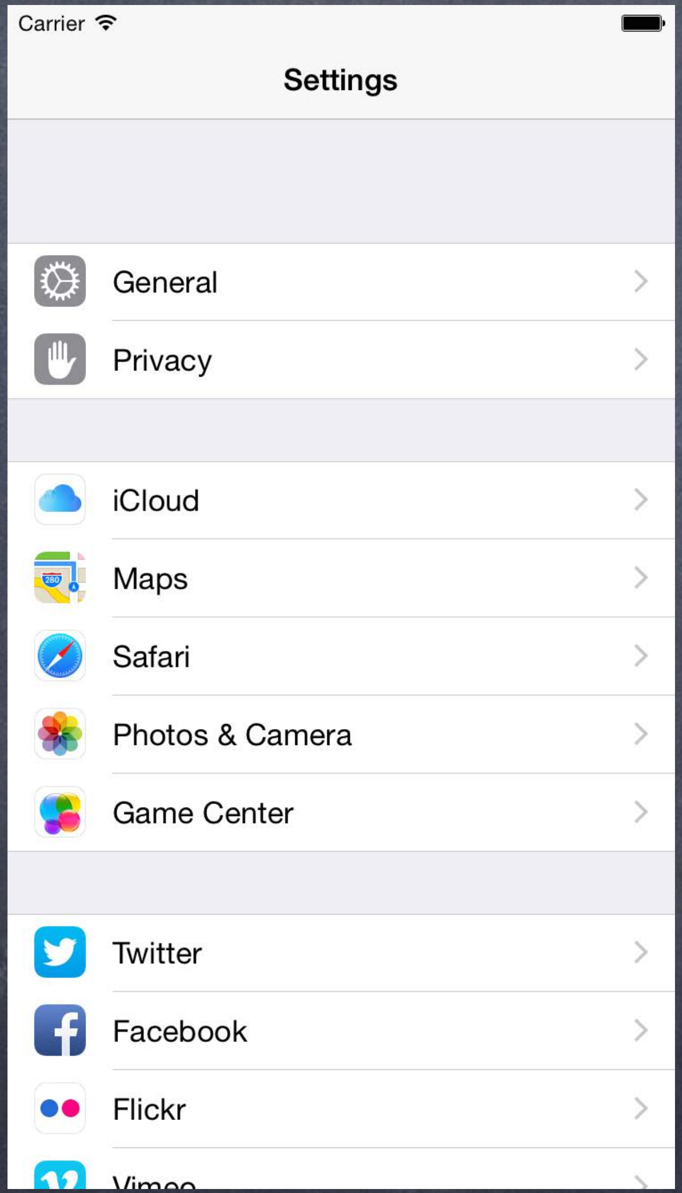
UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …

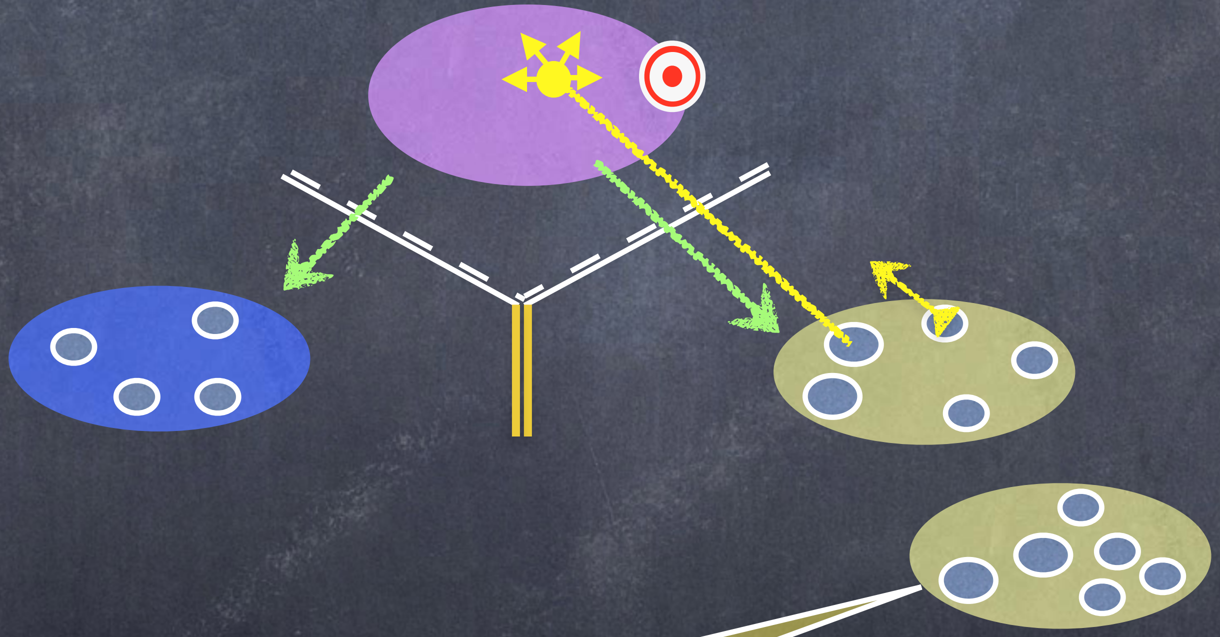


UINavigationController

- Pushes and pops MVCs off of a stack (like a stack of cards) …

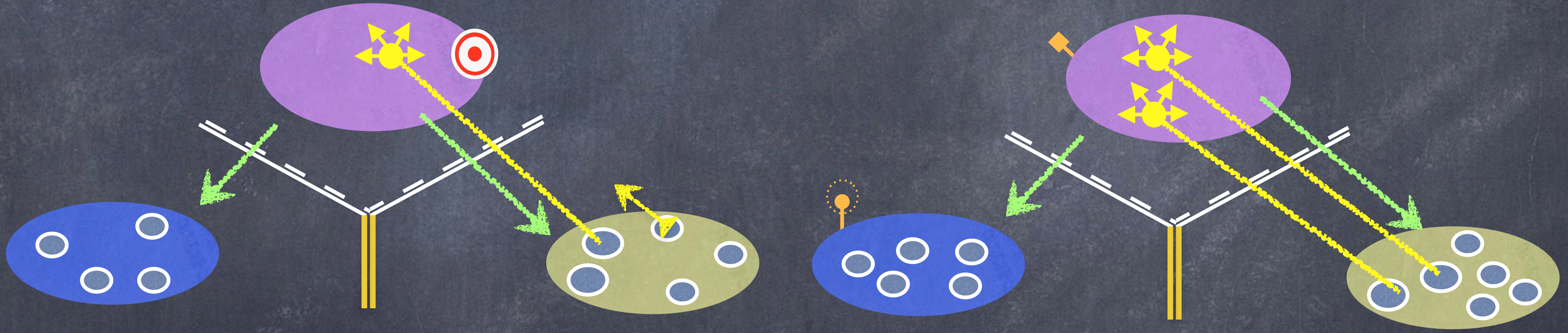


UINavigationController



I want more features, but it doesn't make sense to put them all in one MVC!

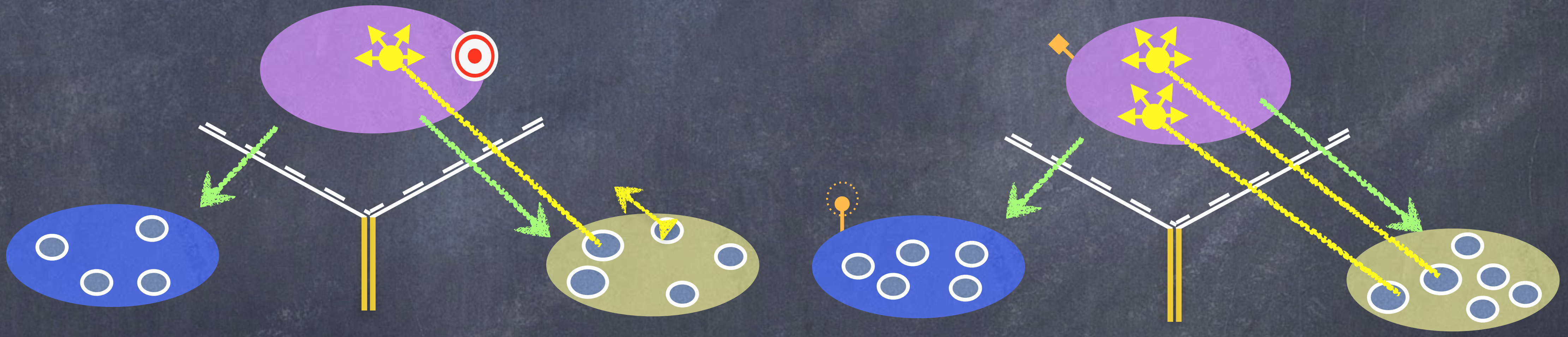
UINavigationController



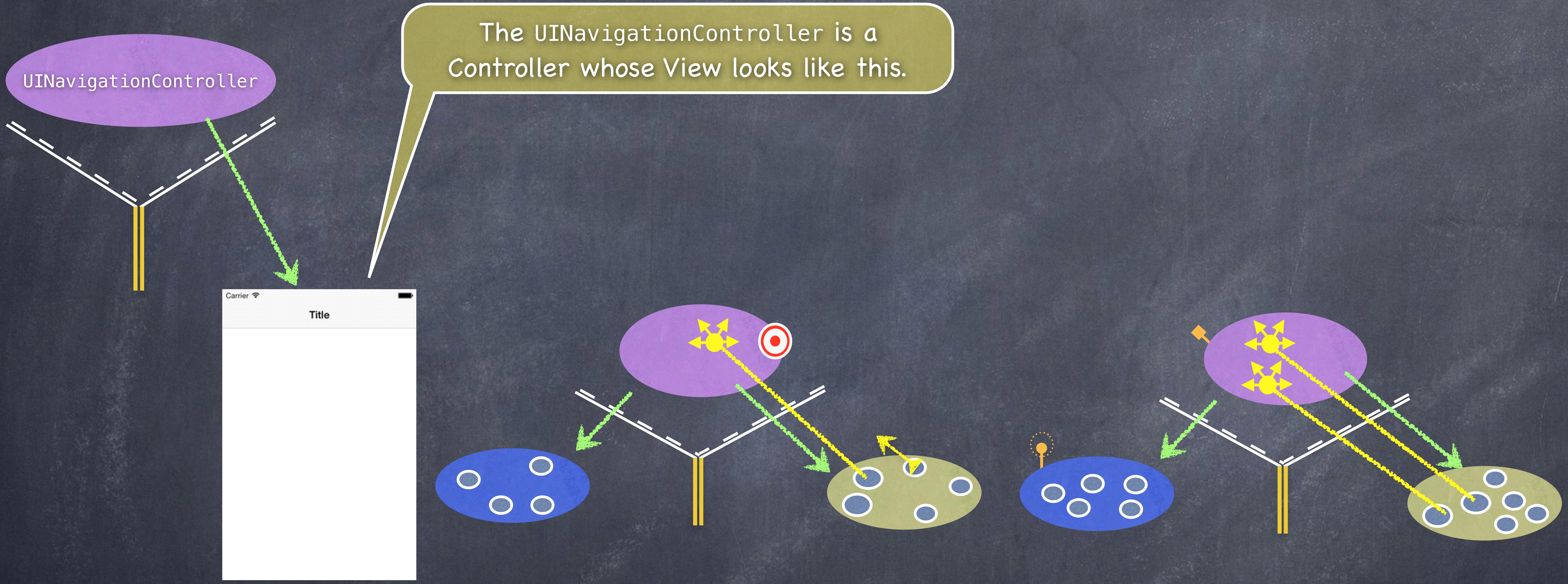
So I create a new MVC to encapsulate that functionality.

UINavigationController

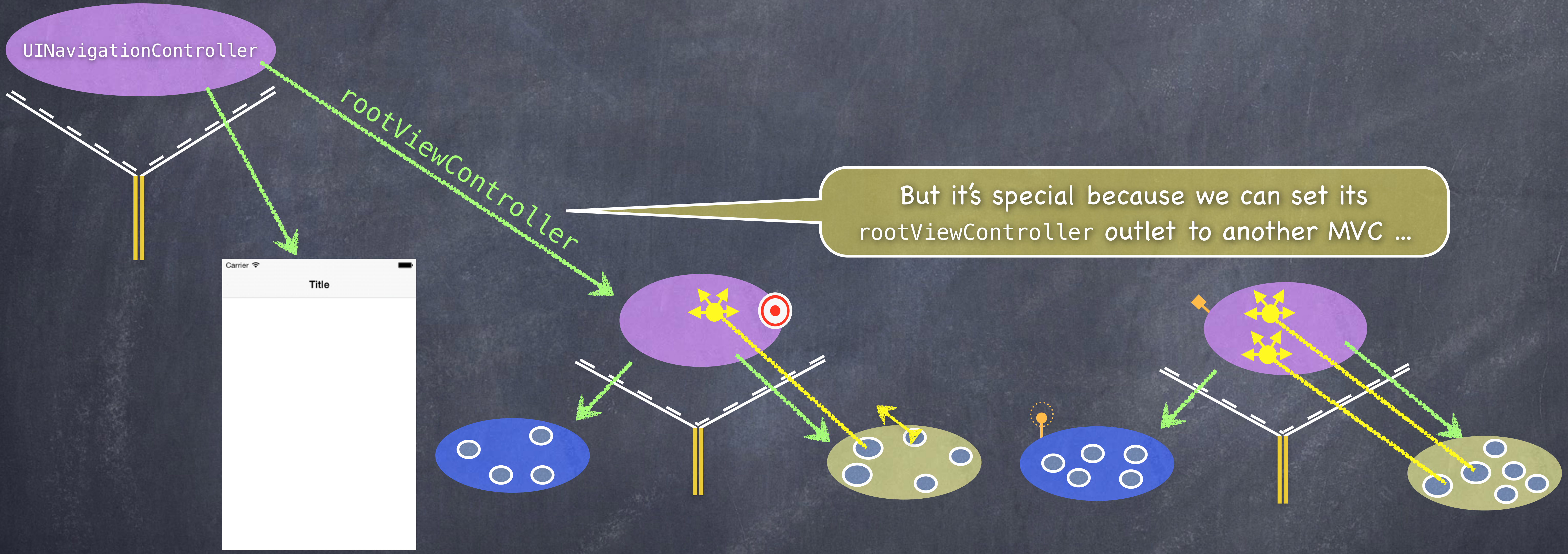
We can use a UINavigationController to let them share the screen.



UINavigationController

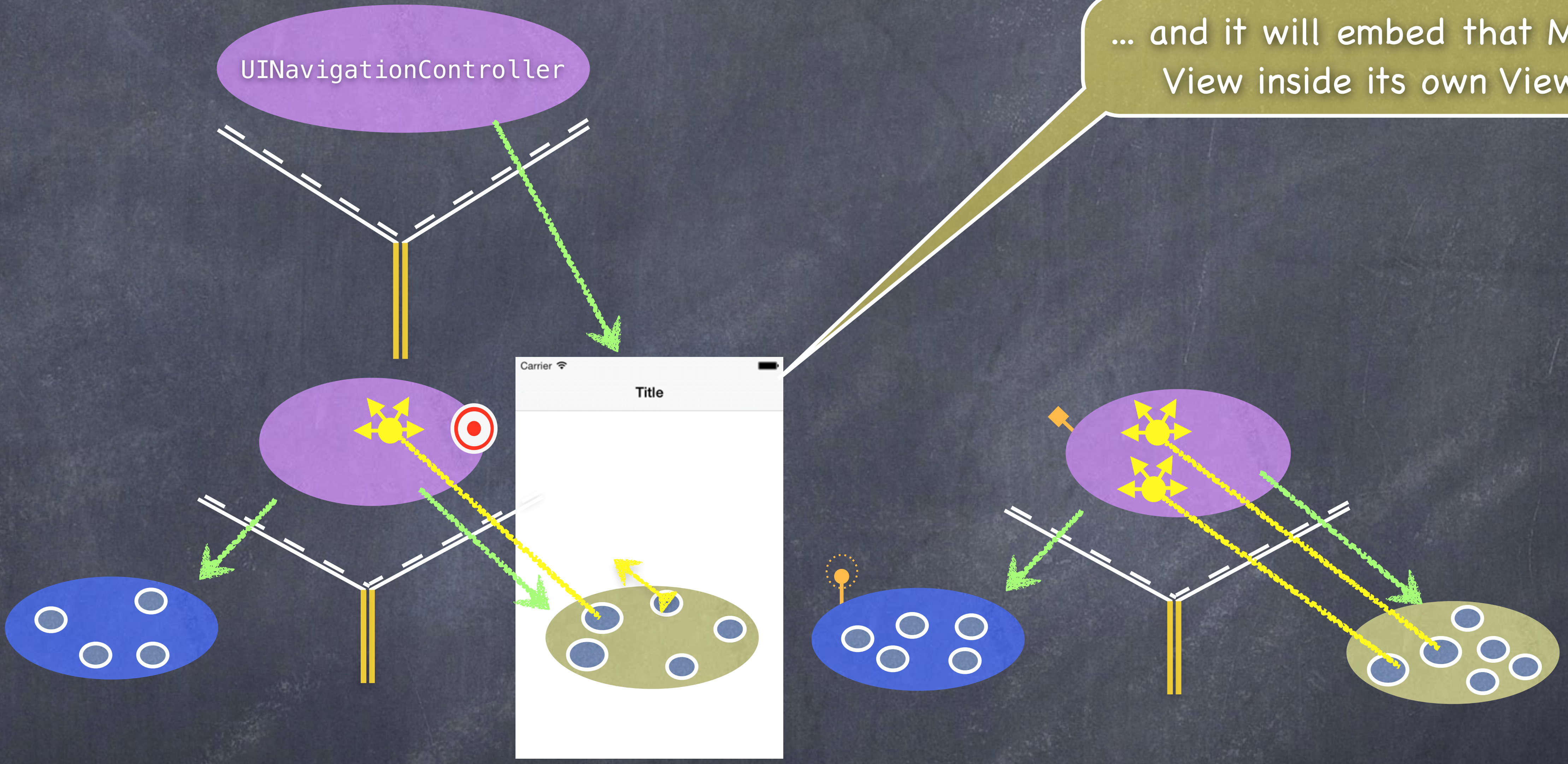

UINavigationController
The UINavigationController is a Controller whose View looks like this.
Carrier
Title

UINavigationController



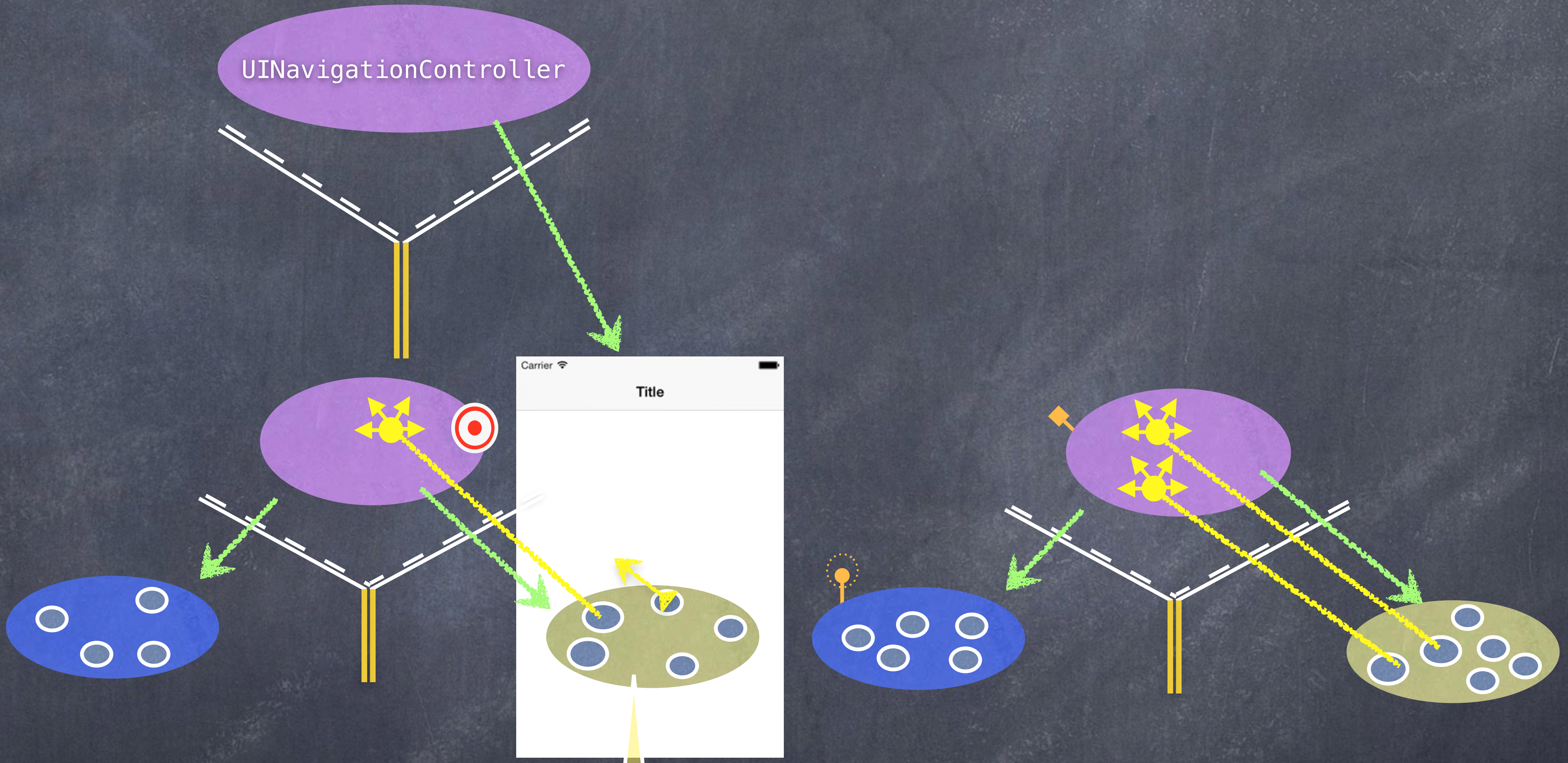
But it's special because we can set its rootViewController outlet to another MVC ...

UINavigationController



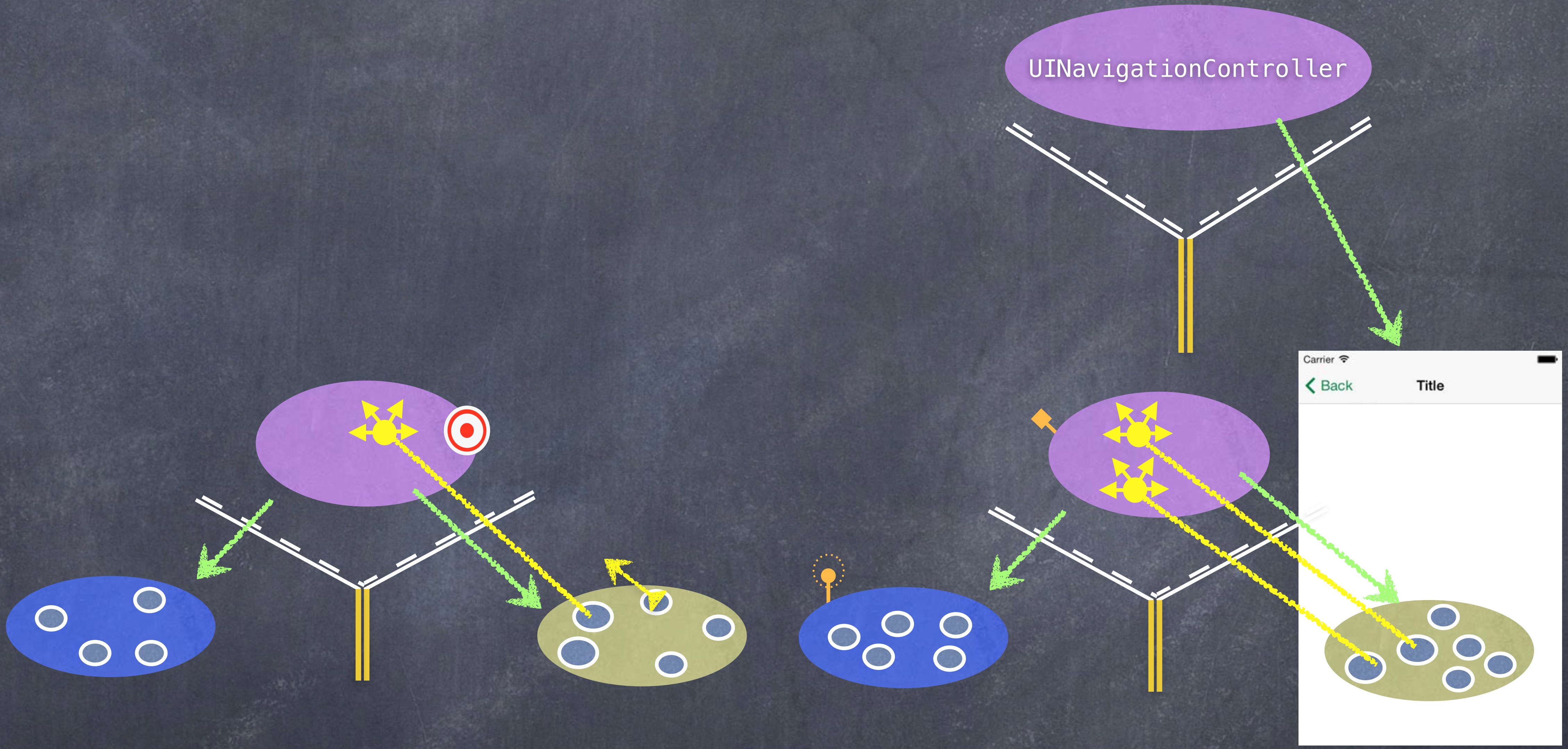
… and it will embed that MVC's View inside its own View.

UINavigationController



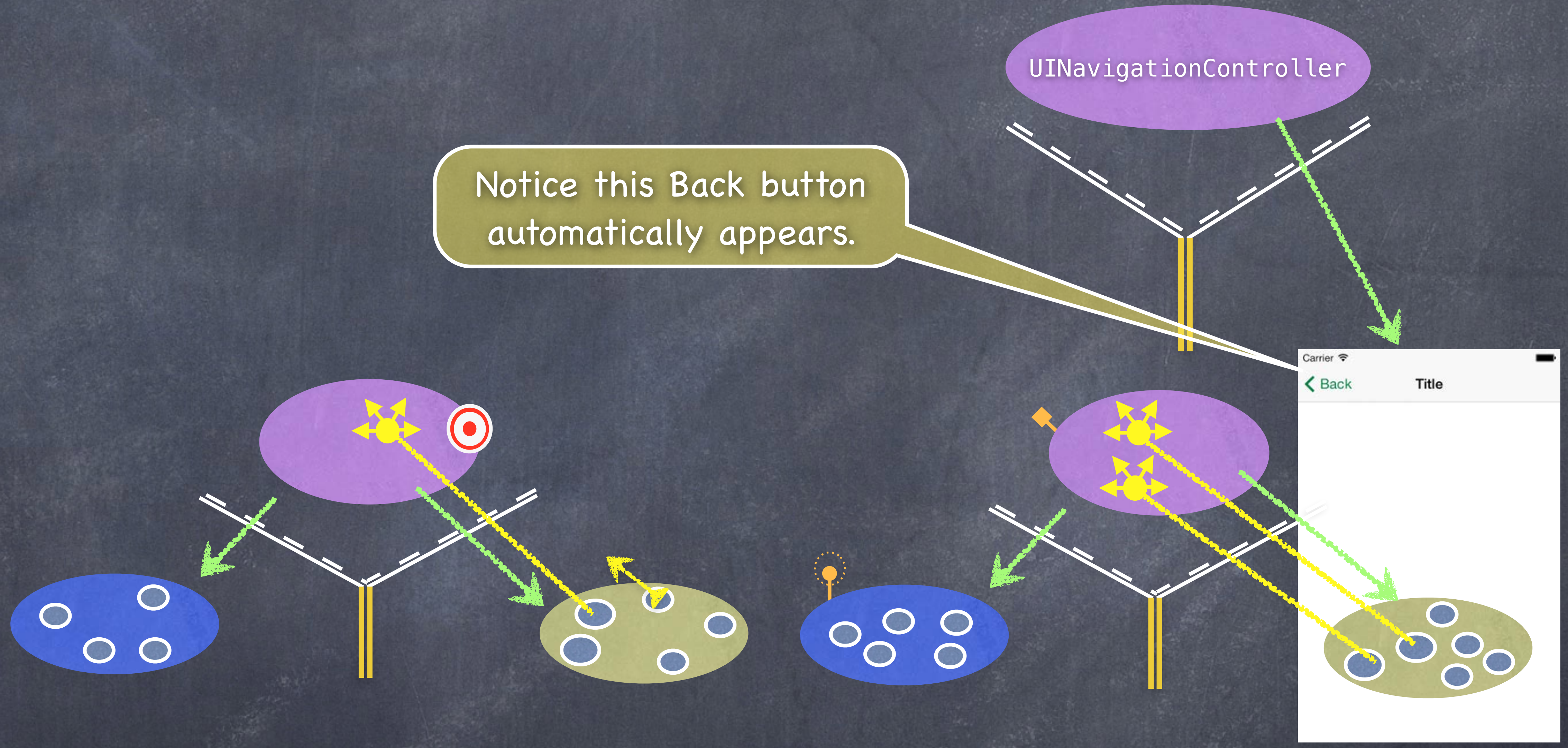
Then a UI element in this View (e.g. a `UIButton`) can <u>segue</u> to the other MVC and its View will now appear in the `UINavigationController` instead.

UINavigationController



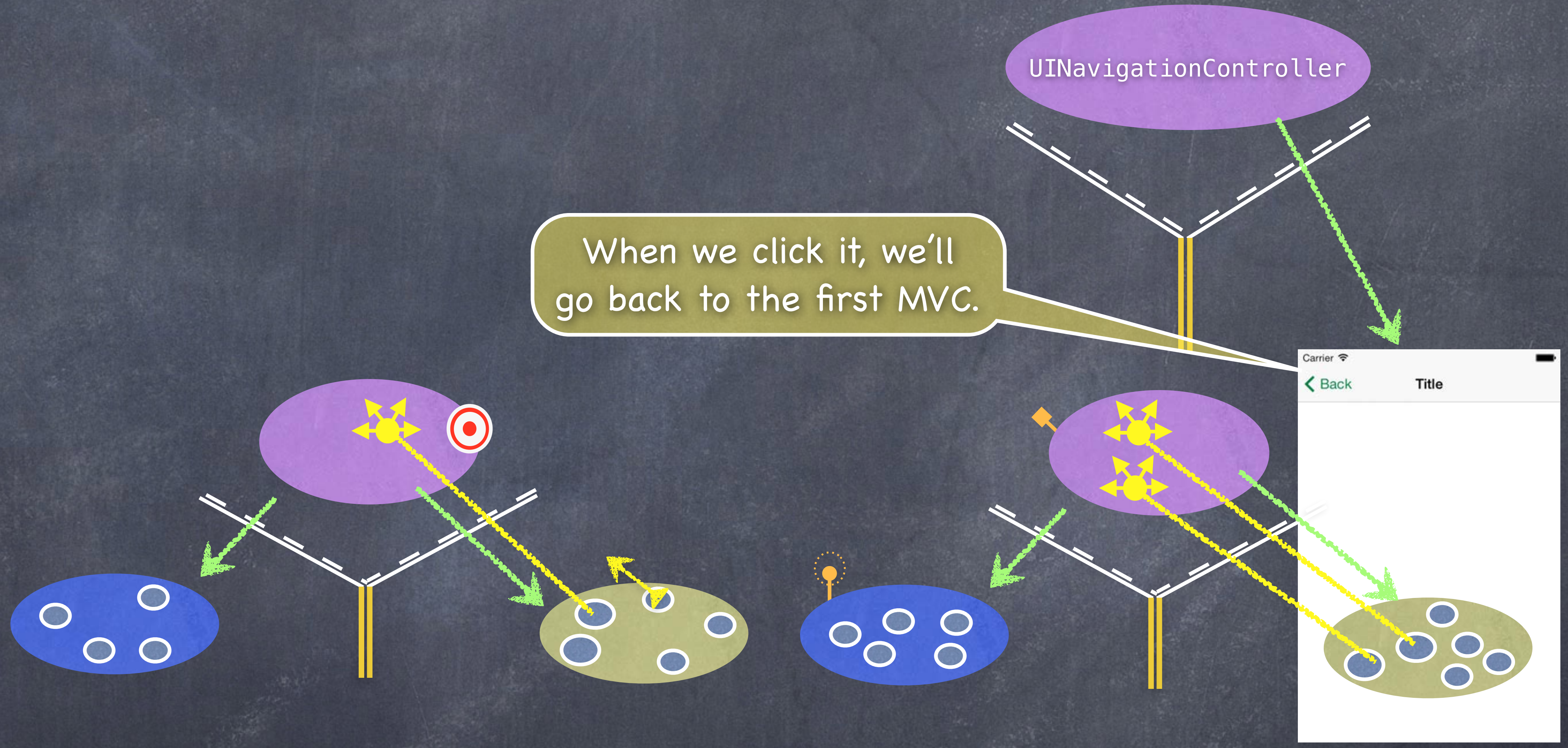
We call this kind of segue a "Show (push) segue".

UINavigationController

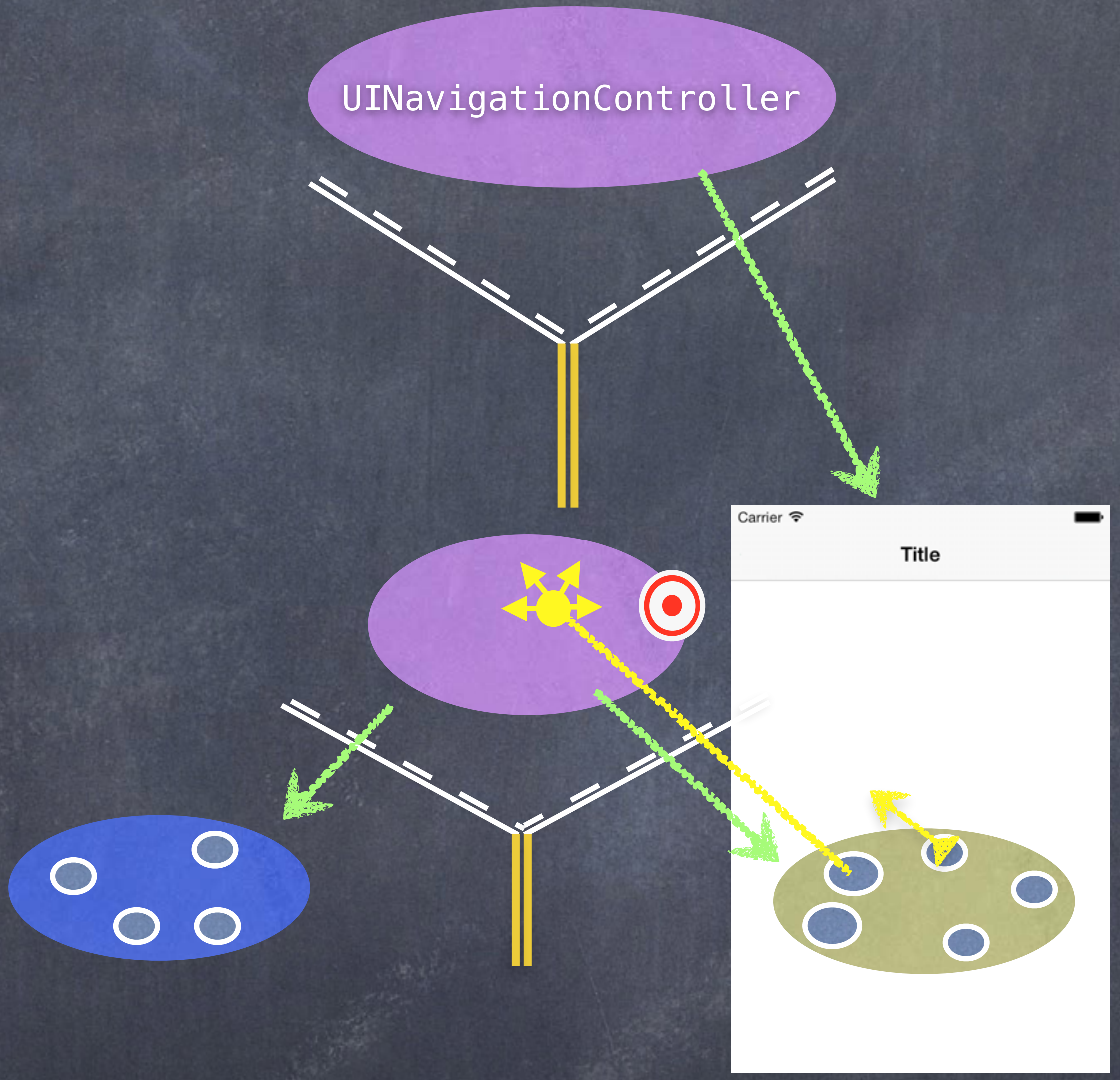


Notice this Back button automatically appears.

UINavigationController



UINavigationController



Notice that after we back out of an MVC,
it disappears (it is deallocated from the heap, in fact).

Accessing the sub-MVCs

- You can get the sub-MVCs via the viewControllers property

```
var viewControllers: [UIViewController]? { get set } // can be optional (e.g. for tab bar)
// for a tab bar, they are in order, left to right, in the array
// for a split view, [0] is the master and [1] is the detail
// for a navigation controller, [0] is the root and the rest are in order on the stack
// even though this is settable, usually setting happens via storyboard, segues, or other
// for example, navigation controller's push and pop methods
```

- But how do you get ahold of the SVC, TBC or NC itself?

Every UIViewController knows the Split View, Tab Bar or Navigation Controller it is currently in

These are UIViewController properties …

```
var tabBarController: UITabBarController? { get }
var splitViewController: UISplitViewController? { get }
var navigationController: UINavigationController? { get }
```

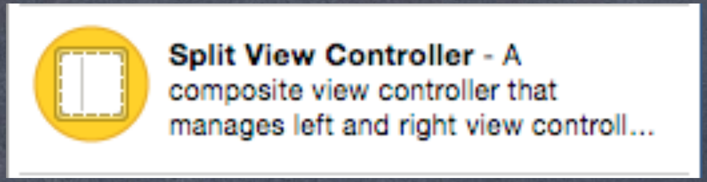
So, for example, to get the detail (right side) of the split view controller you are in …

```
if let detail: UIViewController? = splitViewController?.viewControllers[1] { … }
```

Wiring up MVCs

- How do we wire all this stuff up?

  Let’s say we have a Calculator MVC and a Calculator Graphing MVC
  How do we hook them up to be the two sides of a Split View?

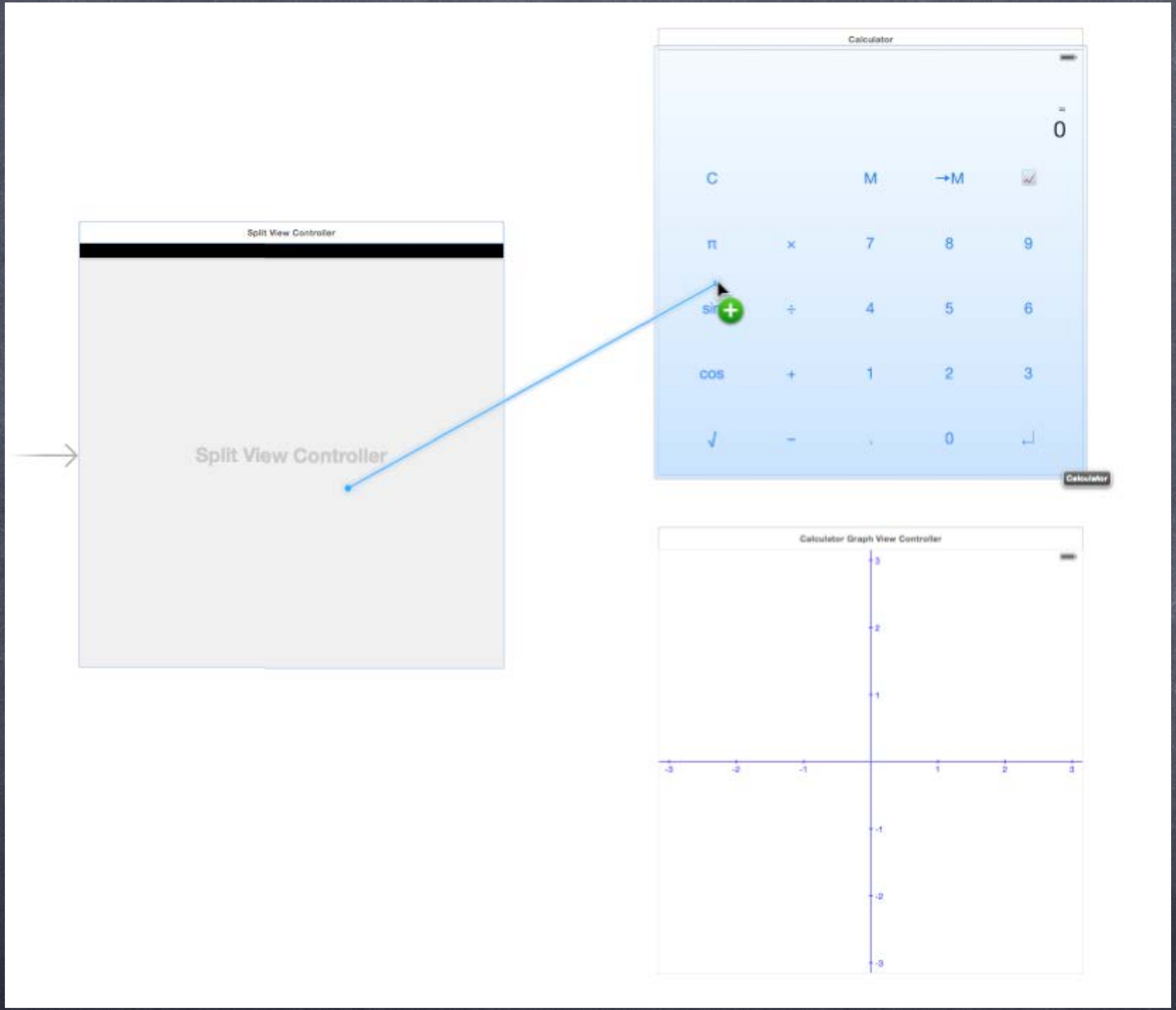
  Just drag out a 

  Split View Controller - A composite view controller that manages left and right view controll...

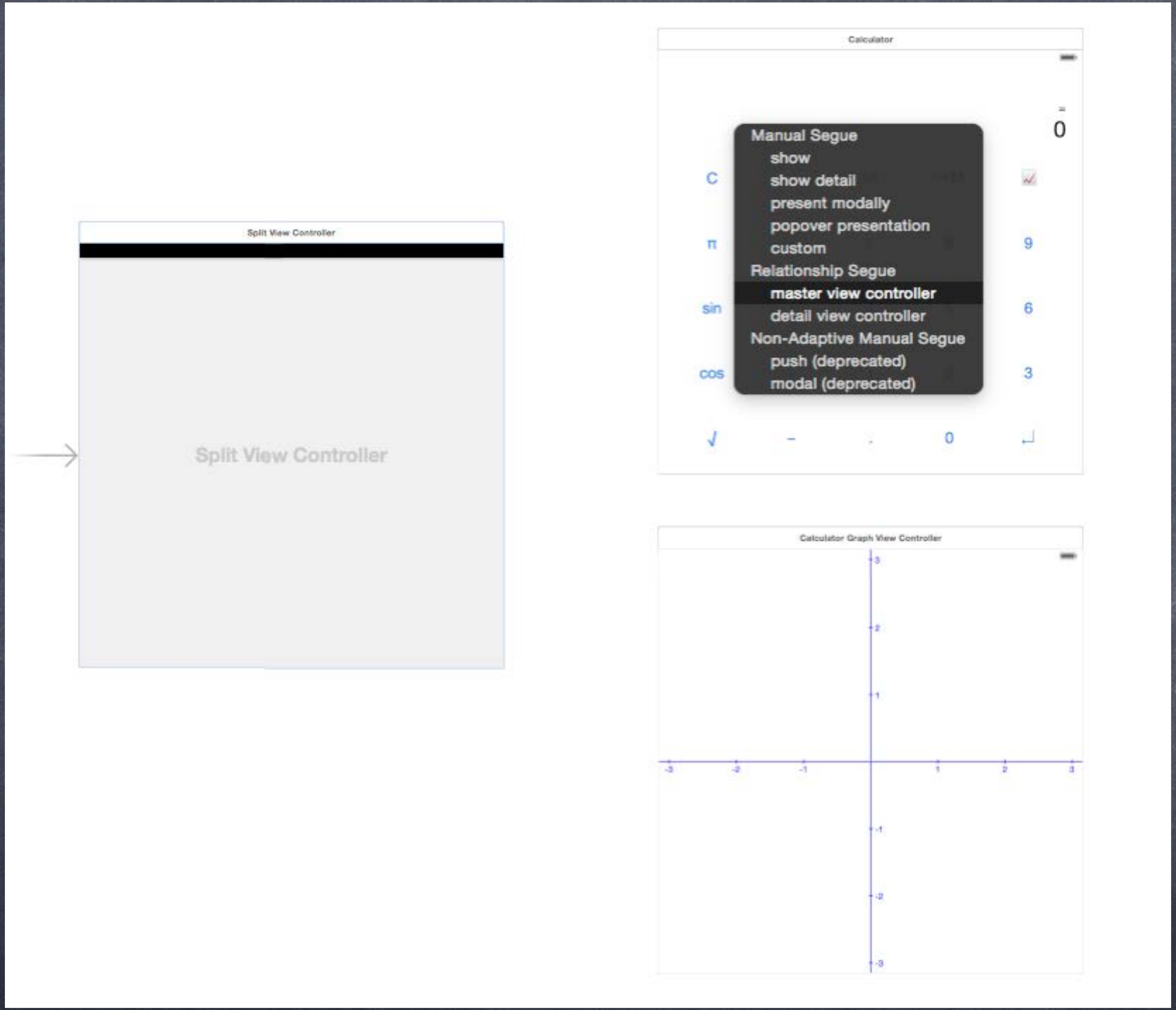
  (and delete all the extra VCs it brings with it)

  Then ctrl-drag from the `UISplitViewController` to the master and detail MVCs …

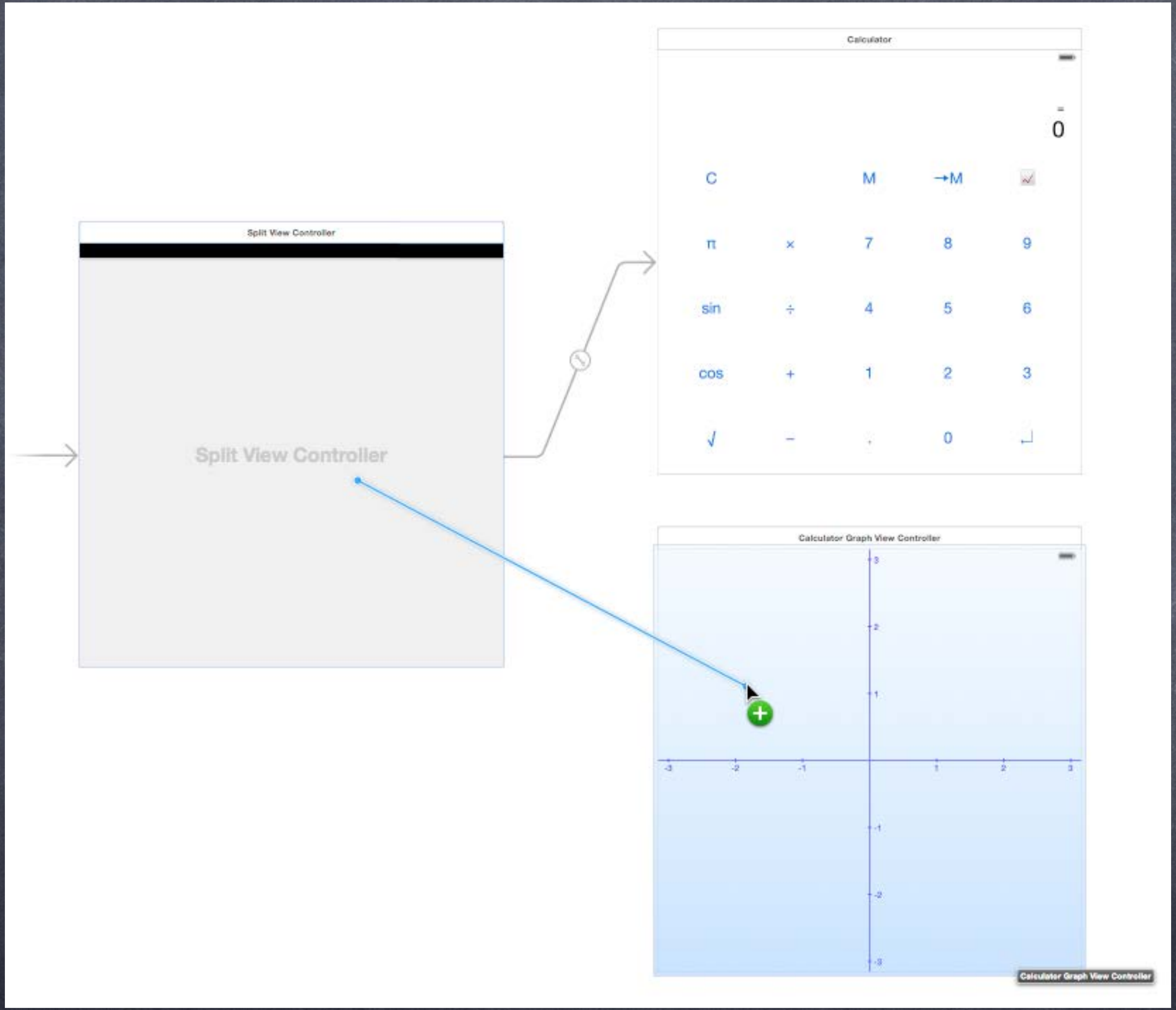
Wiring up MVCs



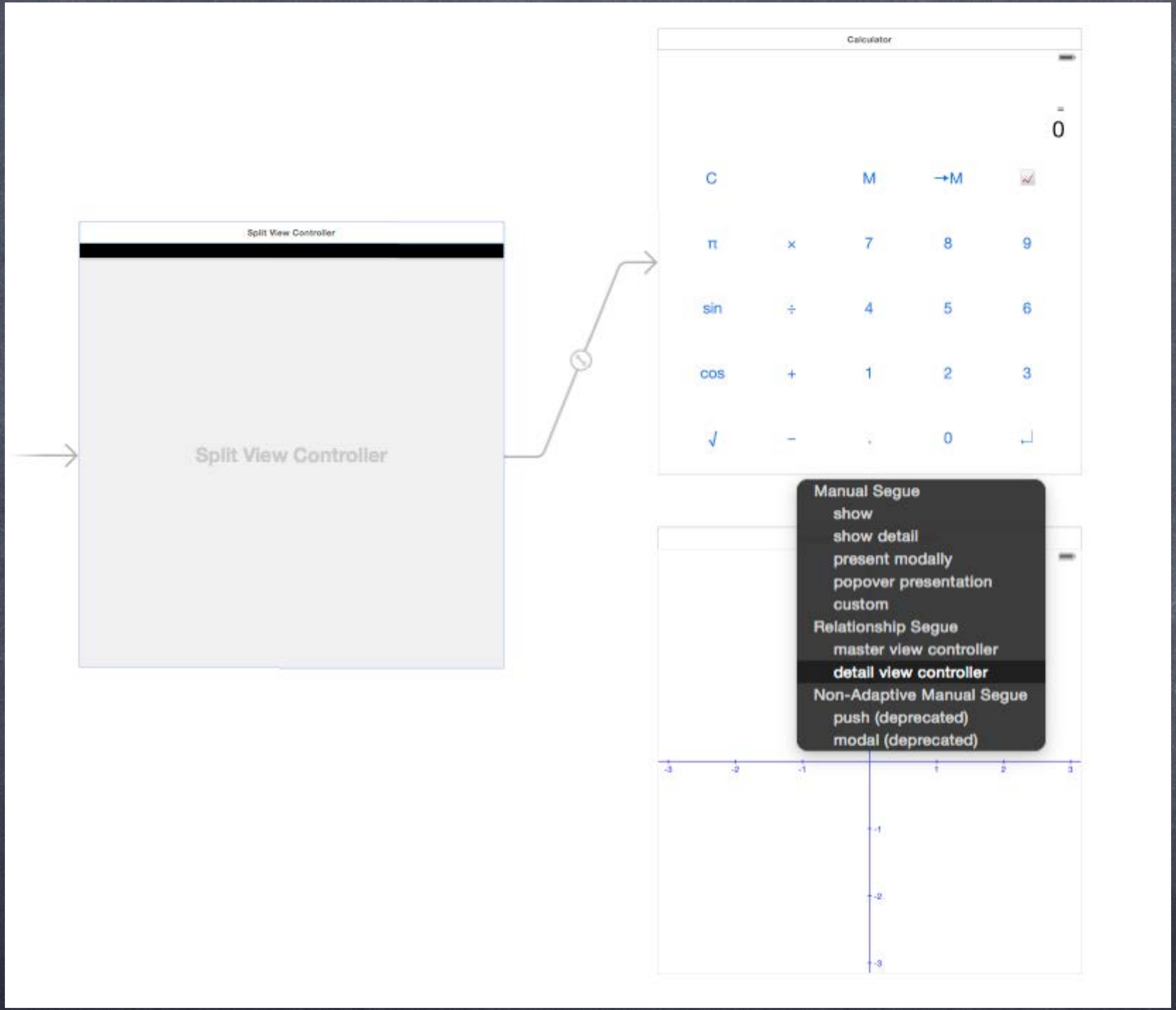
Wiring up MVCs



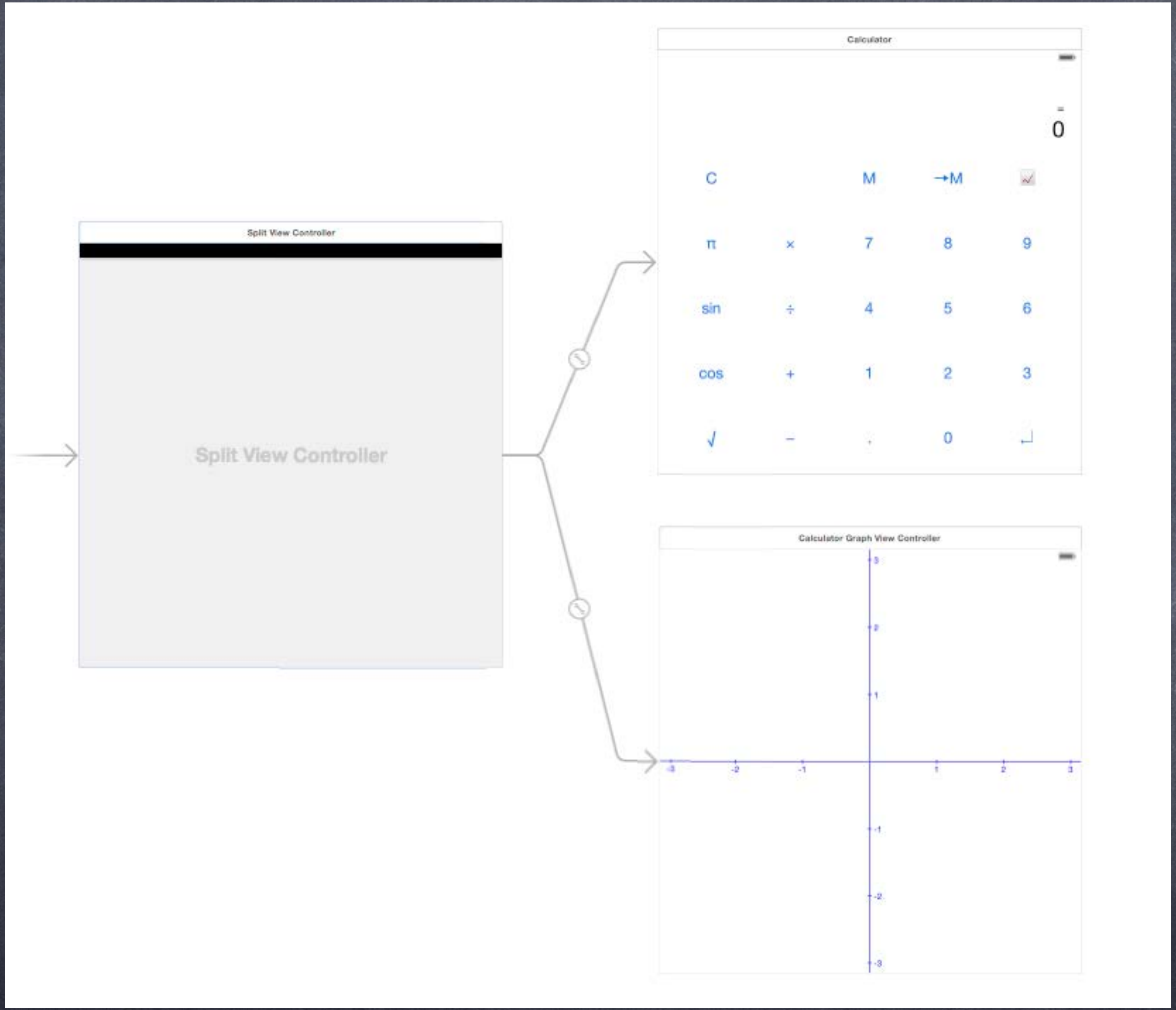
Wiring up MVCs



Wiring up MVCs

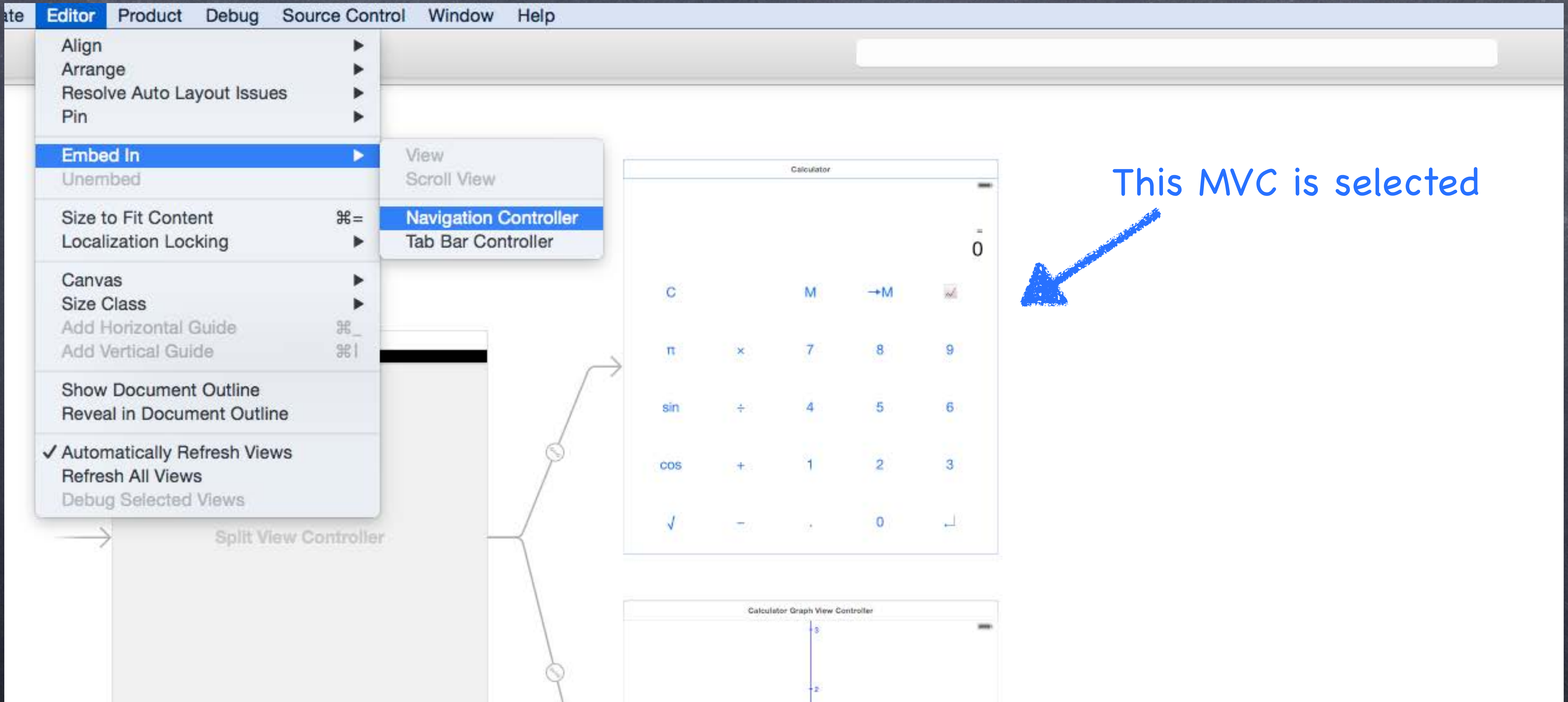


Wiring up MVCs



Wiring up MVCs

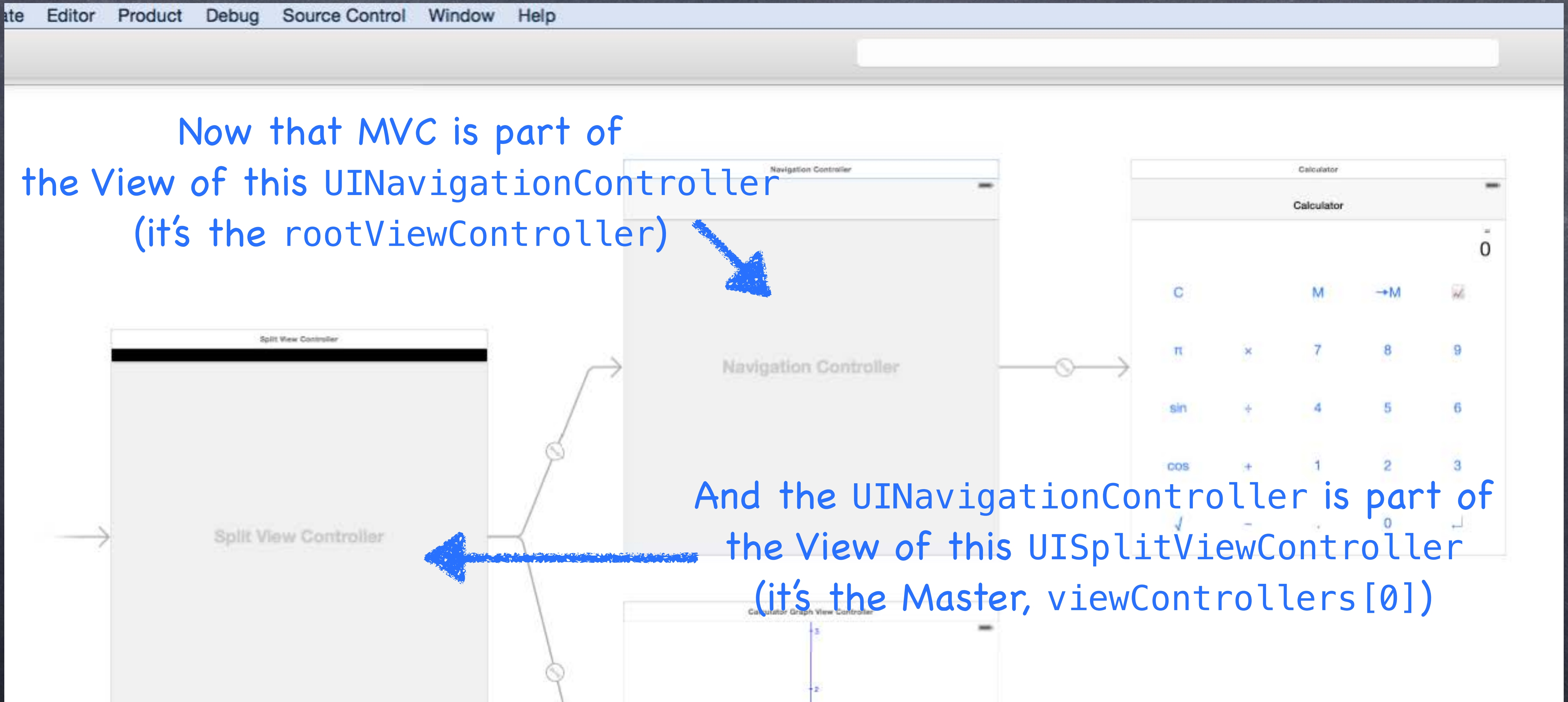
- But split view can only do its thing properly on iPad/iPhone+
  - So we need to put some Navigation Controllers in there so it will work on iPhone
  - The Navigation Controllers will be good for iPad too because the MVCs will get titles
  - The simplest way to wrap a Navigation Controller around an MVC is with Editor->Embed In



This MVC is selected

Wiring up MVCs

- But split view can only do its thing properly on iPad/iPhone+
  - So we need to put some Navigation Controllers in there so it will work on iPhone
  - The Navigation Controllers will be good for iPad too because the MVCs will get titles
  - The simplest way to wrap a Navigation Controller around an MVC is with Editor->Embed In



ate Editor Product Debug Source Control Window Help
Navigation Controller
Calculator
Split View Controller
Calculator Graph View Controller

Now that MVC is part of the View of this UINavigationController (it's the rootViewController)

And the UINavigationController is part of the View of this UISplitViewController (it's the Master, viewControllers[0])

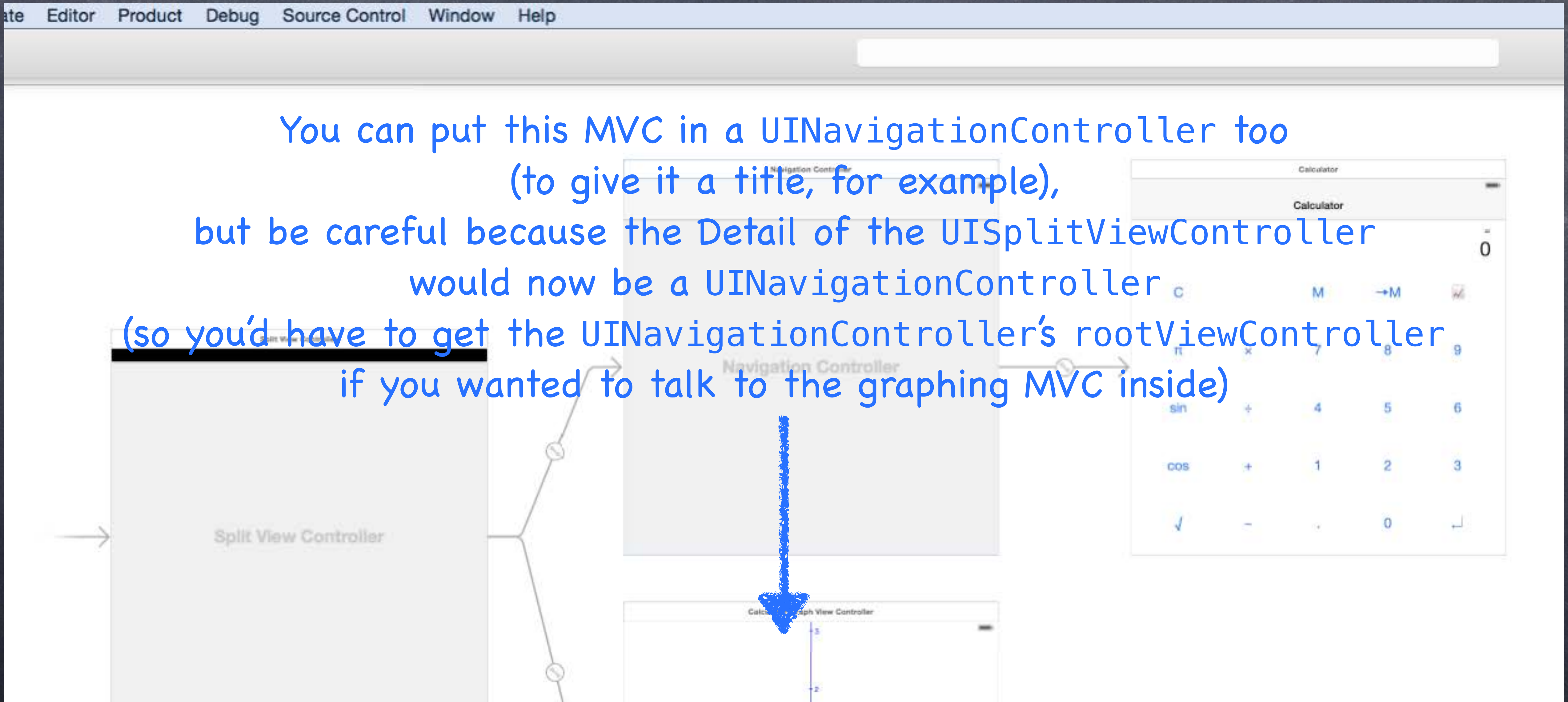
Wiring up MVCs

- But split view can only do its thing properly on iPad/iPhone+

  So we need to put some Navigation Controllers in there so it will work on iPhone

  The Navigation Controllers will be good for iPad too because the MVCs will get titles

  The simplest way to wrap a Navigation Controller around an MVC is with Editor->Embed In



You can put this MVC in a UINavigationController too
(to give it a title, for example),
but be careful because the Detail of the UISplitViewController
would now be a UINavigationController
(so you'd have to get the UINavigationController's rootViewController
if you wanted to talk to the graphing MVC inside)

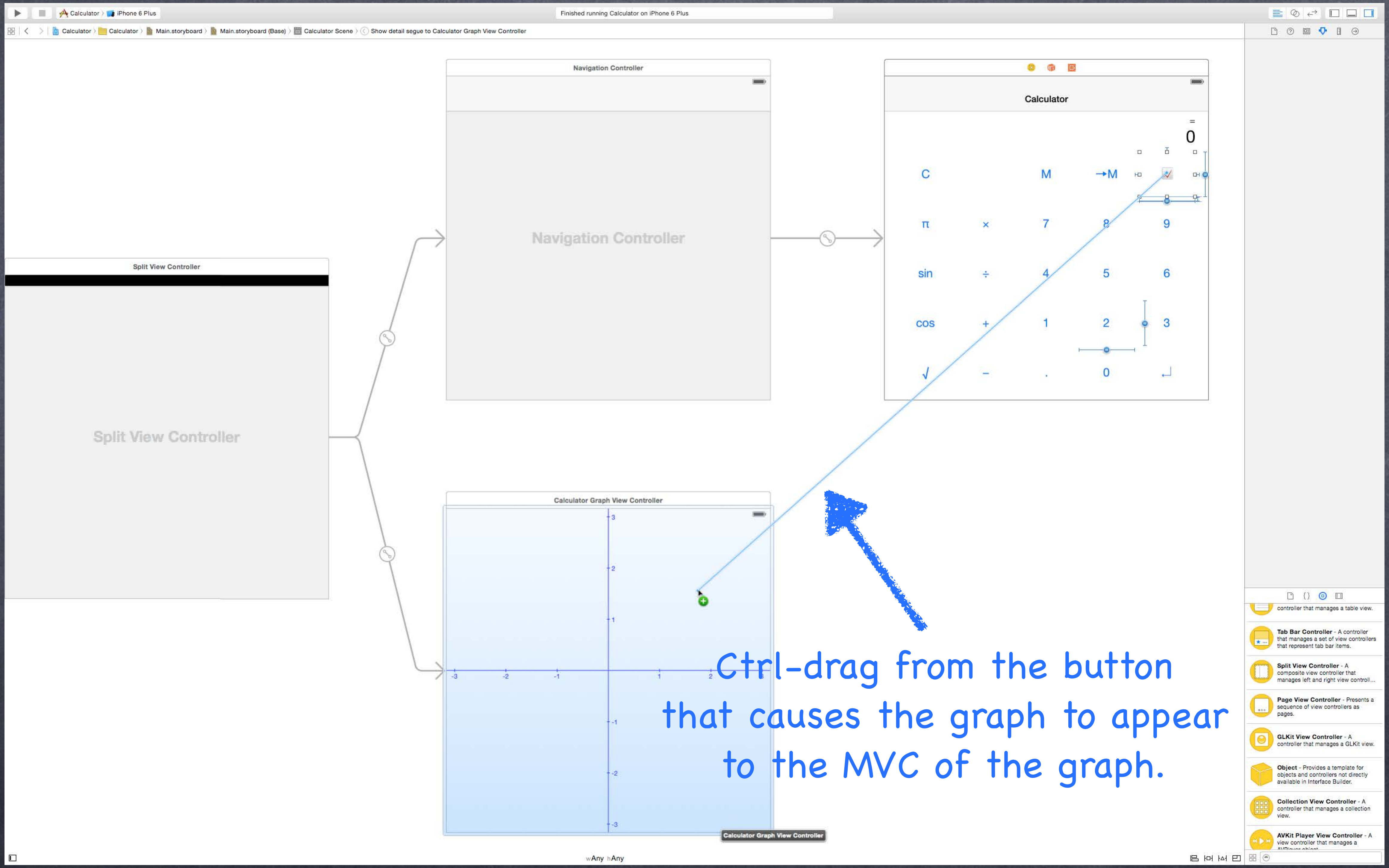
Segues

- We've built up our Controllers of Controllers, now what?
  - Now we need to make it so that one MVC can cause another to appear
  - We call that a "segue"
- Kinds of segues (they will adapt to their environment)
  - Show Segue (will push in a Navigation Controller, else Modal)
  - Show Detail Segue (will show in Detail of a Split View or will push in a Navigation Controller)
  - Modal Segue (take over the entire screen while the MVC is up)
  - Popover Segue (make the MVC appear in a little popover window)
- Segues always create a new instance of an MVC
  - This is important to understand
  - Even the Detail of a Split View will get replaced with a new instance of that MVC
  - When you segue in a Navigation Controller it will not segue to some old instance, it'll be new
  - Going "back" in a Navigation Controller is NOT a segue though (so no new instance there)

Segues

- How do we make these segues happen?
  - Ctrl-drag in a storyboard from an instigator (like a button) to the MVC to segue to
  - Can be done in code as well

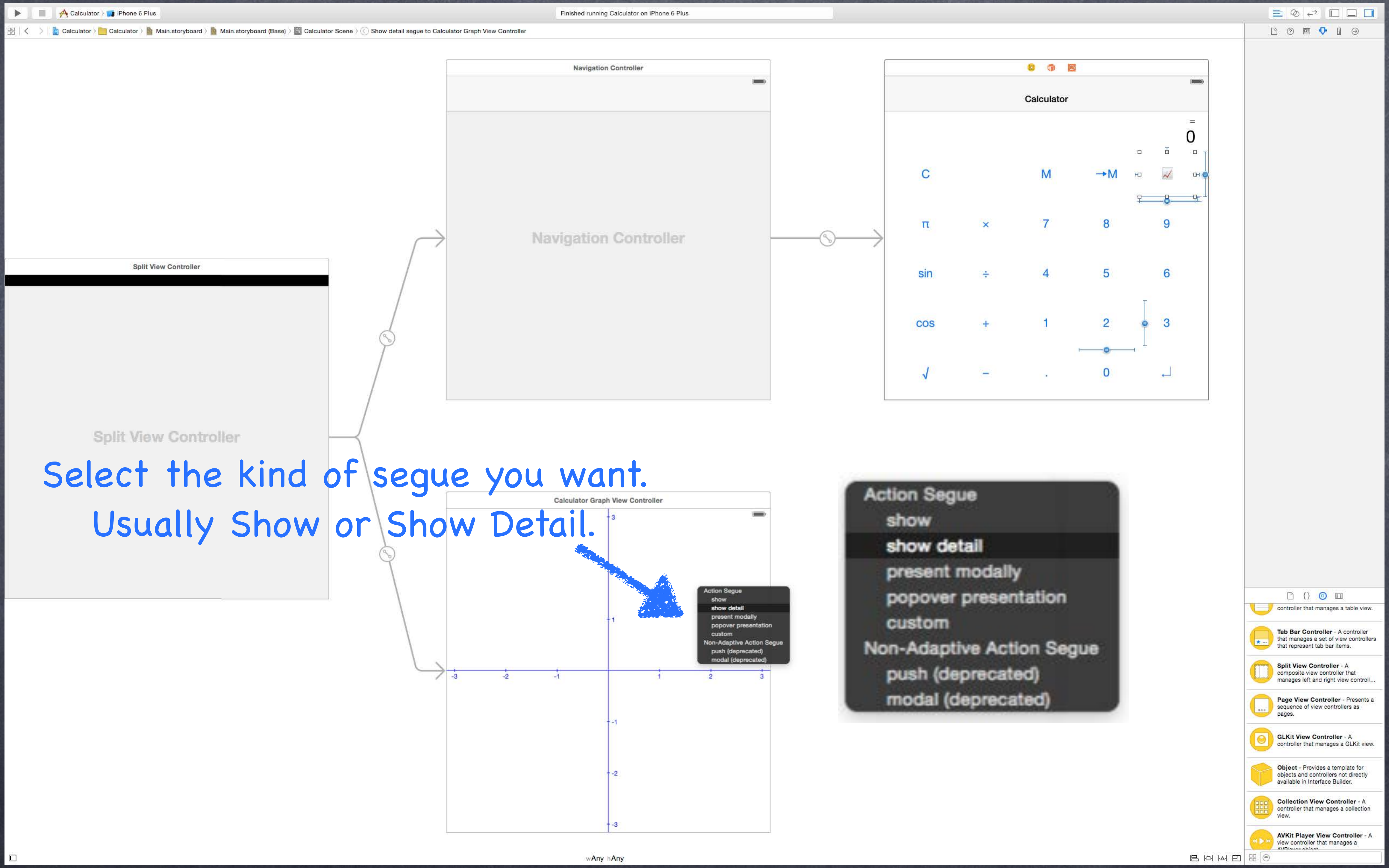
Segues



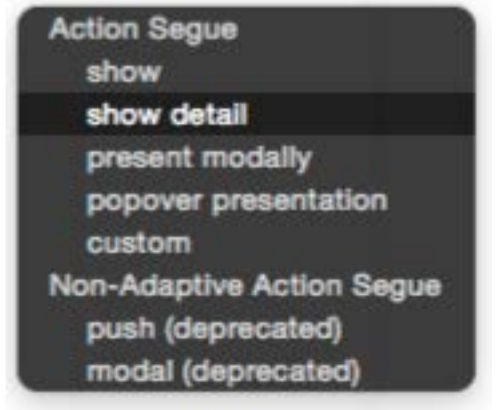
Calculator › iPhone 6 Plus
Finished running Calculator on iPhone 6 Plus
Calculator › Calculator › Main.storyboard › Main.storyboard (Base) › Calculator Scene › Show detail segue to Calculator Graph View Controller
Split View Controller
Navigation Controller
Calculator
Calculator Graph View Controller

Ctrl-drag from the button that causes the graph to appear to the MVC of the graph.

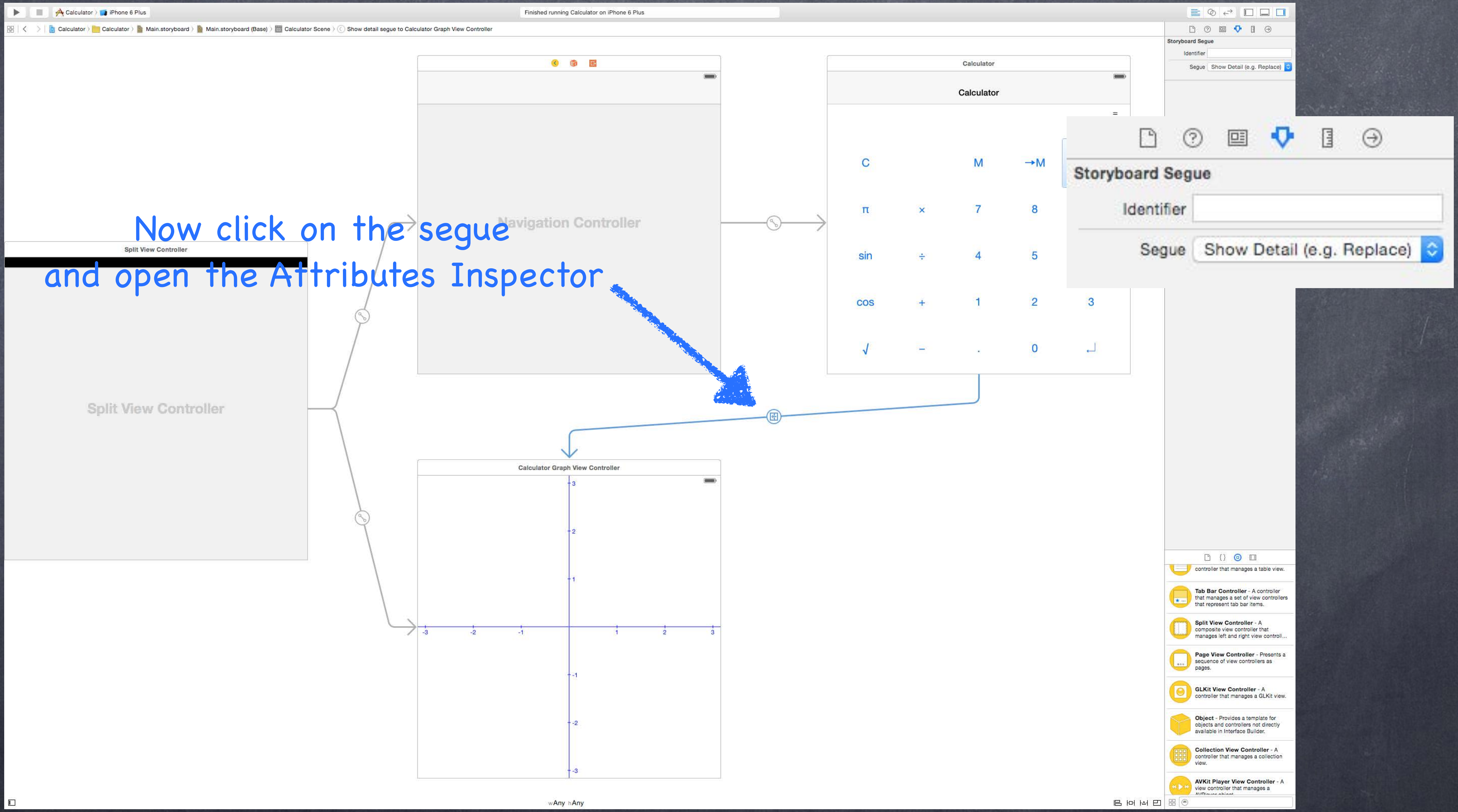
Segues



Select the kind of segue you want.
Usually Show or Show Detail.



Segues



Now click on the segue
and open the Attributes Inspector

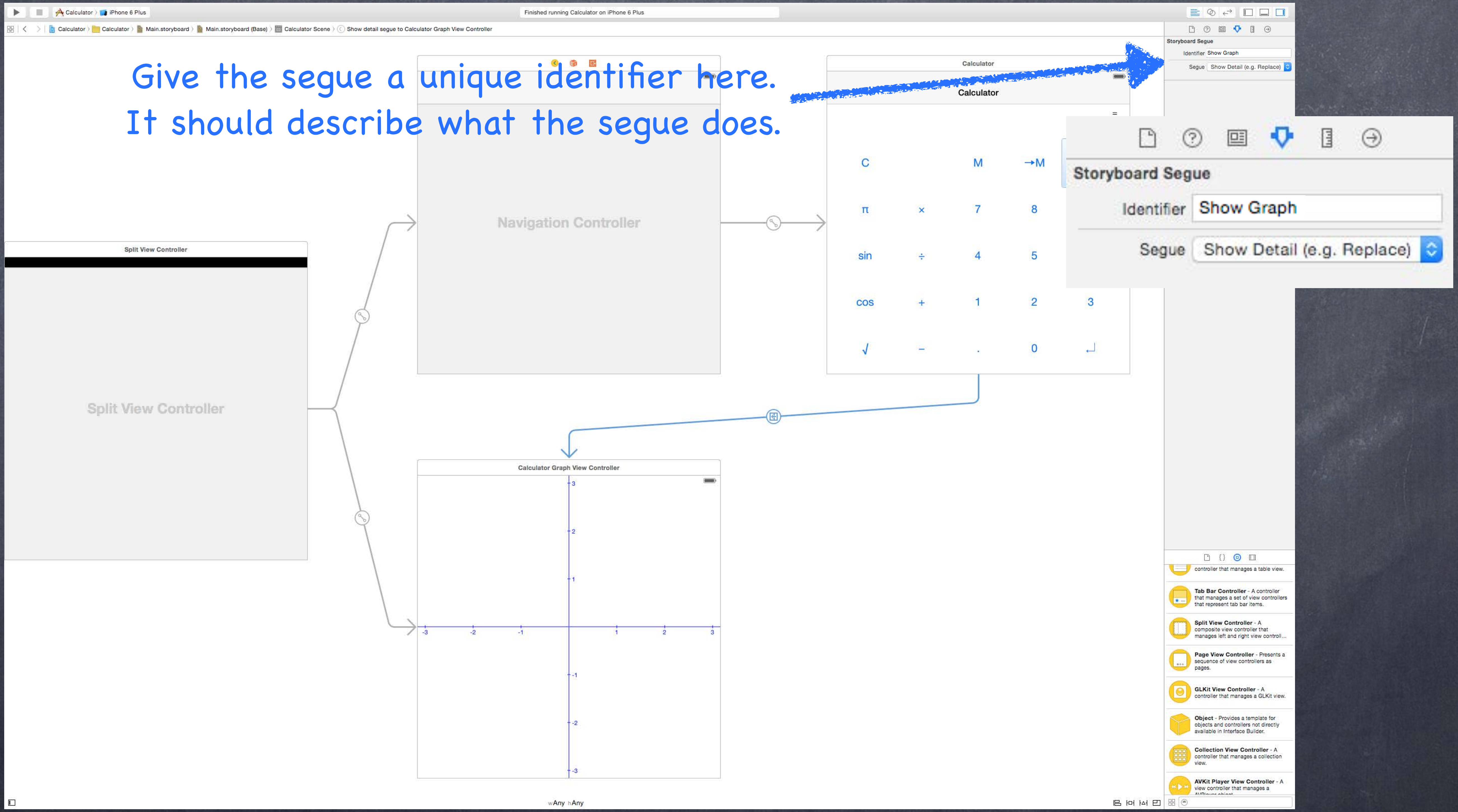
Storyboard Segue

Identifier

Segue: Show Detail (e.g. Replace)

Calculator
iPhone 6 Plus
Finished running Calculator on iPhone 6 Plus
Calculator › Calculator › Main.storyboard › Main.storyboard (Base) › Calculator Scene › Show detail segue to Calculator Graph View Controller
Split View Controller
Navigation Controller
Calculator
Calculator Graph View Controller
C
M
→M
π
×
7
8
sin
÷
4
5
cos
+
1
2
3
√
−
.
0
Tab Bar Controller - A controller that manages a set of view controllers that represent tab bar items.
Split View Controller - A composite view controller that manages left and right view controll...
Page View Controller - Presents a sequence of view controllers as pages.
GLKit View Controller - A controller that manages a GLKit view.
Object - Provides a template for objects and controllers not directly available in Interface Builder.
Collection View Controller - A controller that manages a collection view.
AVKit Player View Controller - A view controller that manages a
wAny hAny

Segues



Give the segue a unique identifier here.
It should describe what the segue does.

Storyboard Segue

Identifier Show Graph

Segue Show Detail (e.g. Replace)

Segues

- What's that identifier all about?
  - You would need it to invoke this segue from code using this `UIViewController` method
  - `func performSegue(withIdentifier: String, sender: Any?)`
  - (but we almost never do this because we set usually ctrl-drag from the instigator)
  - The `sender` can be whatever you want (you'll see where it shows up in a moment)
  - You can ctrl-drag from the Controller itself to another Controller if you're segueing via code
  - (because in that case, you'll be specifying the `sender` above)
- More important use of the identifier: preparing for a segue
  - When a segue happens, the View Controller containing the instigator gets a chance to prepare the destination View Controller to be segued to
  - Usually this means setting up the segued-to MVC's Model and display characteristics
  - Remember that the MVC segued to is always a fresh instance (never a reused one)

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case “Show Graph”:
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case “Show Graph”:
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

The segue passed in contains important information about this segue:

1. the identifier from the storyboard
2. the Controller of the MVC you are segueing to (which was just created for you)

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case "Show Graph":
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

The sender is either the instigating object from a storyboard (e.g. a UIButton)
or the sender you provided (see last slide) if you invoked the segue manually in code

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case "Show Graph":
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

Here is the identifier from the storyboard (it can be `nil`, so be sure to check for that case)
Your Controller might support preparing for lots of different segues from different instigators
so this identifier is how you'll know which one you're preparing for

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case "Show Graph":
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

For this example, we'll assume we entered "Show Graph" in the Attributes Inspector when we had the segue selected in the storyboard

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case "Show Graph":
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

Here we are looking at the Controller of the MVC we're segueing to
It is Any so we must cast it to the Controller we (should) know it to be

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case "Show Graph":
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

This is where the actual preparation of the segued-to MVC occurs
Hopefully the MVC has a clear public API that it wants you to use to prepare it
Once the MVC is prepared, it should run on its own power (only using delegation to talk back)

Preparing for a Segue

- The method that is called in the instigator's Controller

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
            case "Show Graph":
                if let vc = segue.destinationViewController as? GraphController {
                    vc.property1 = …
                    vc.callMethodToSetItUp(…)
                }
            default: break
        }
    }
}
```

It is crucial to understand that this preparation is happening BEFORE outlets get set!
It is a very common bug to prepare an MVC thinking its outlets are set.

Preventing Segues

- You can prevent a segue from happening too

  Just return `false` from this method your `UIViewController` ...

  ```
  func shouldPerformSegue(withIdentifier identifier: String?, sender: Any?) -> Bool
  ```

  The `identifier` is the one in the storyboard.

  The `sender` is the instigating object (e.g. the button that is causing the segue).