Stanford CS193p

Developing Applications for iOS
Spring 2016

Today

- Interface Builder Demo
  - Viewing and editing your custom `UIViews` in your storyboard (`FaceView`)
- The `FaceViewController` MVC's Model
  - It's a facial expression
- Gestures
  - Getting touch input from users
- Demo: Modifying the facial expression
  - Panning, pinching, tapping, rotating
- Multiple MVCs
  - Tab Bar, Navigation and Split View Controller

Demo

- Interface Builder Demo

  Viewing and editing your custom `UIViews` in your storyboard (`FaceView`)

- 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 said UIView …

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

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

  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 said UIView …

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

  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 said UIView …

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

  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 (in this case, 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 said UIView …

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

  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 (in this case, the Controller itself)
  The action is the method invoked on recognition ((_:) means the method has an argument)

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 said UIView …

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

  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 (in this case, the Controller itself)
  The action is the method invoked on recognition (`(_:)` means the method has an argument)
  Here we ask the UIView to actually start trying to recognize this gesture in its bounds

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 said UIView …

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

  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 (in this case, the Controller itself)
  The action is the method invoked on recognition ((_:) means the method has an argument)
  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 translationInView(UIView) -> CGPoint` // cumulative since start of recognition
  - `func velocityInView(UIView) -> CGPoint` // how fast the finger is moving (points/s)
  - `func setTranslation(CGPoint, inView: 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 discrete gesture (e.g. a `Swipe`), it changes to `.Recognized` (`Tap` is not a normal discrete)
  - For a continues gesture (e.g. a `Pan`), it moves from `.Began` thru repeated `.Changed` to `.Ended`
  - 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(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(_:)` (if no `_:`, then no gesture argument)

Gestures

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

```
func pan(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(_:)` (if no `_:`, then no gesture argument)

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(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(_:)` (if no _:, then no gesture argument)
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(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(_:)` (if no _:, then no gesture argument)
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(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(_:)` (if no _:, then no gesture argument)
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(gesture: UIPanGestureRecognizer) {
    switch gesture.state {
        case .Changed: fallthrough
        case .Ended:
            let translation = gesture.translationInView(pannableView)
            // update anything that depends on the pan gesture using translation.x and .y
            gesture.setTranslation(CGPointZero, inView: pannableView)
        default: break
    }
}
```

Remember that the action was `pan(_:)` (if no _:, then no gesture argument)
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, then look for `.Recognized`

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

- UITapGestureRecognizer

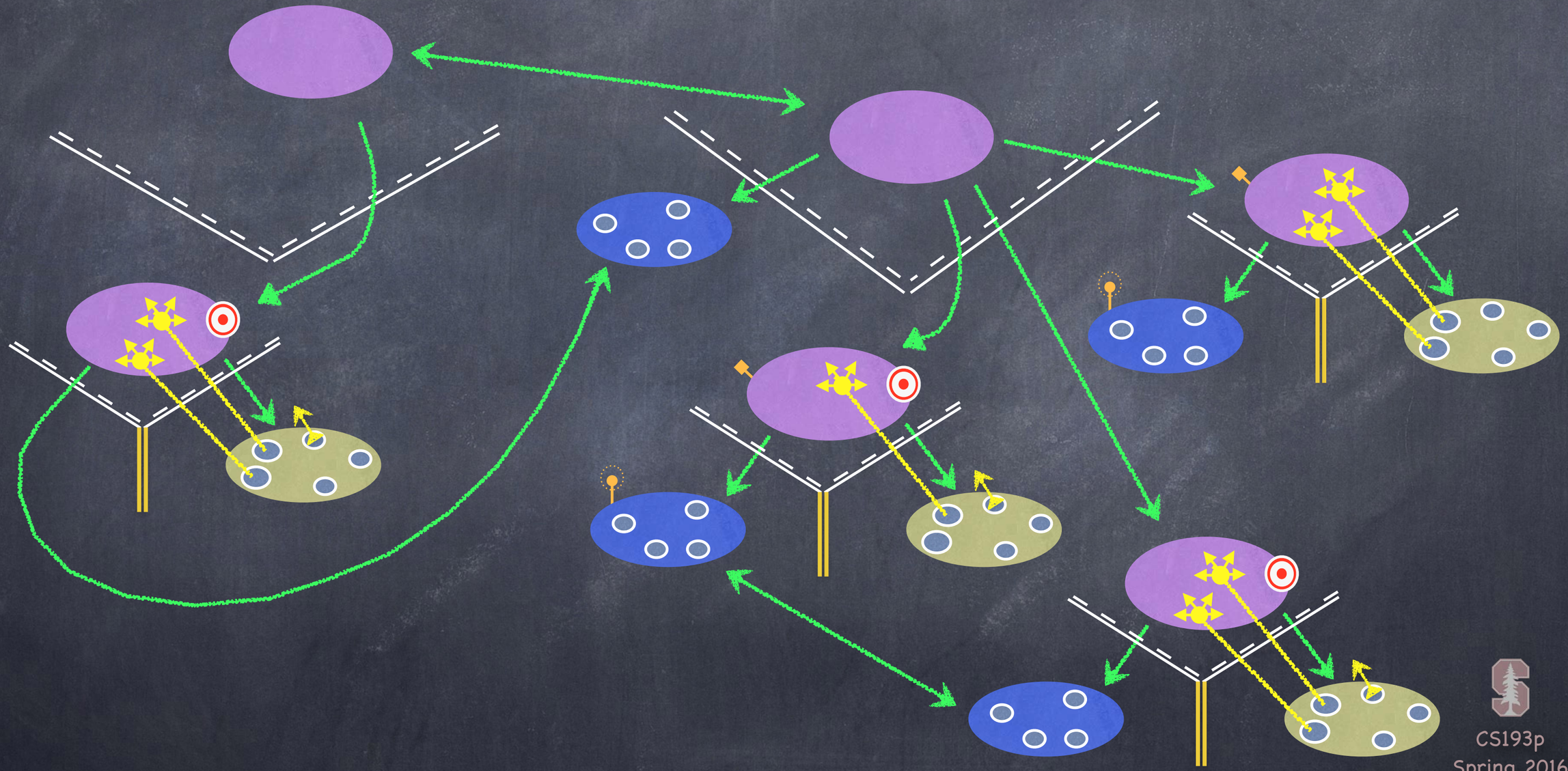
Set up the number of taps and fingers you want, then look for `.Ended`

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

Demo

- Gestures Demo
  - Add a gesture recognizer (pinch) to the `FaceView` to zoom in and out (control its own `scale`)
  - Add a gesture recognizer (pan) to the `FaceView` to control `expression` (Model) in the Controller

MVCs working together

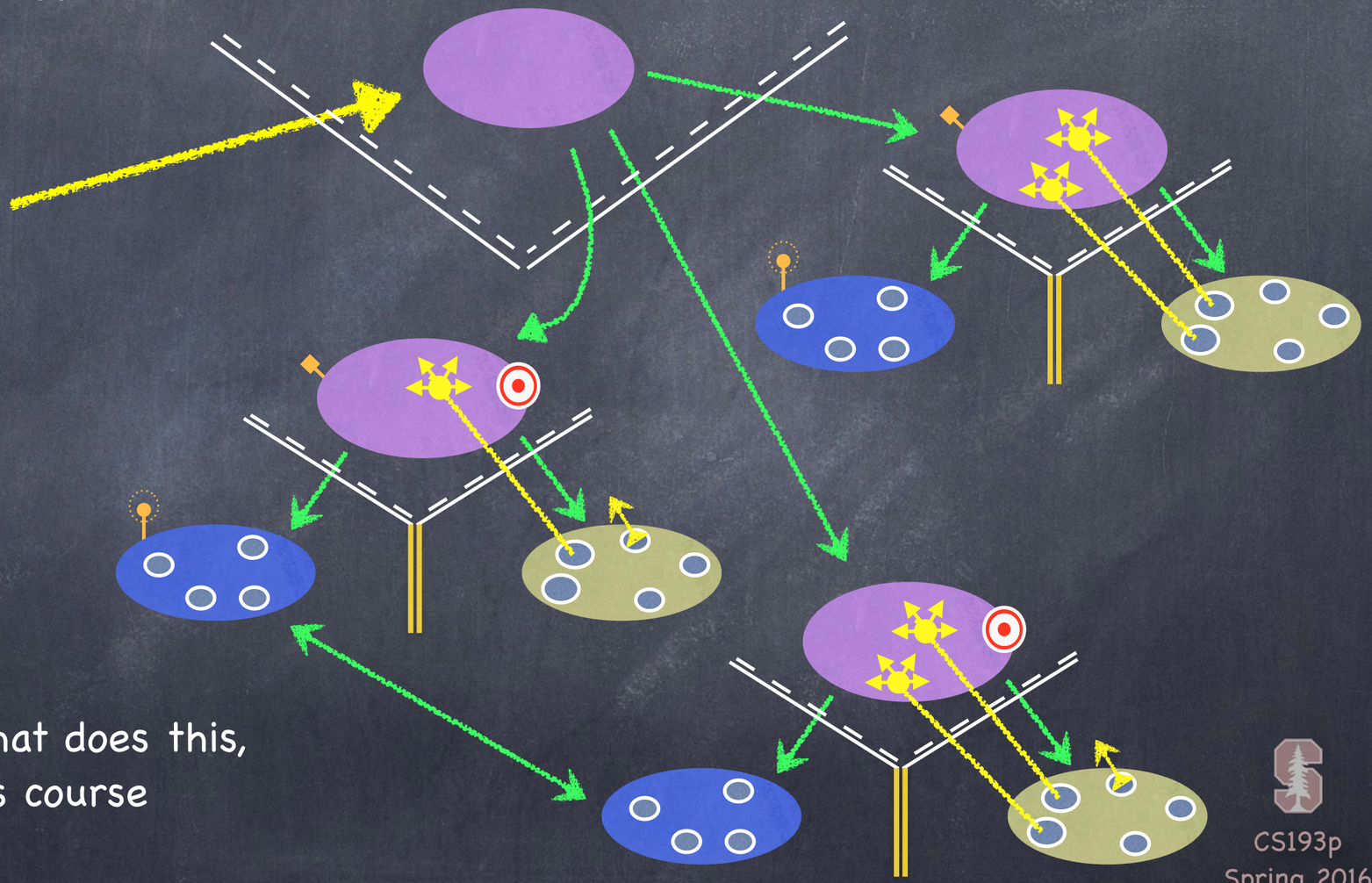


Multiple MVCs

- Time to build more powerful applications

  To do this, we must combine MVCs …

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



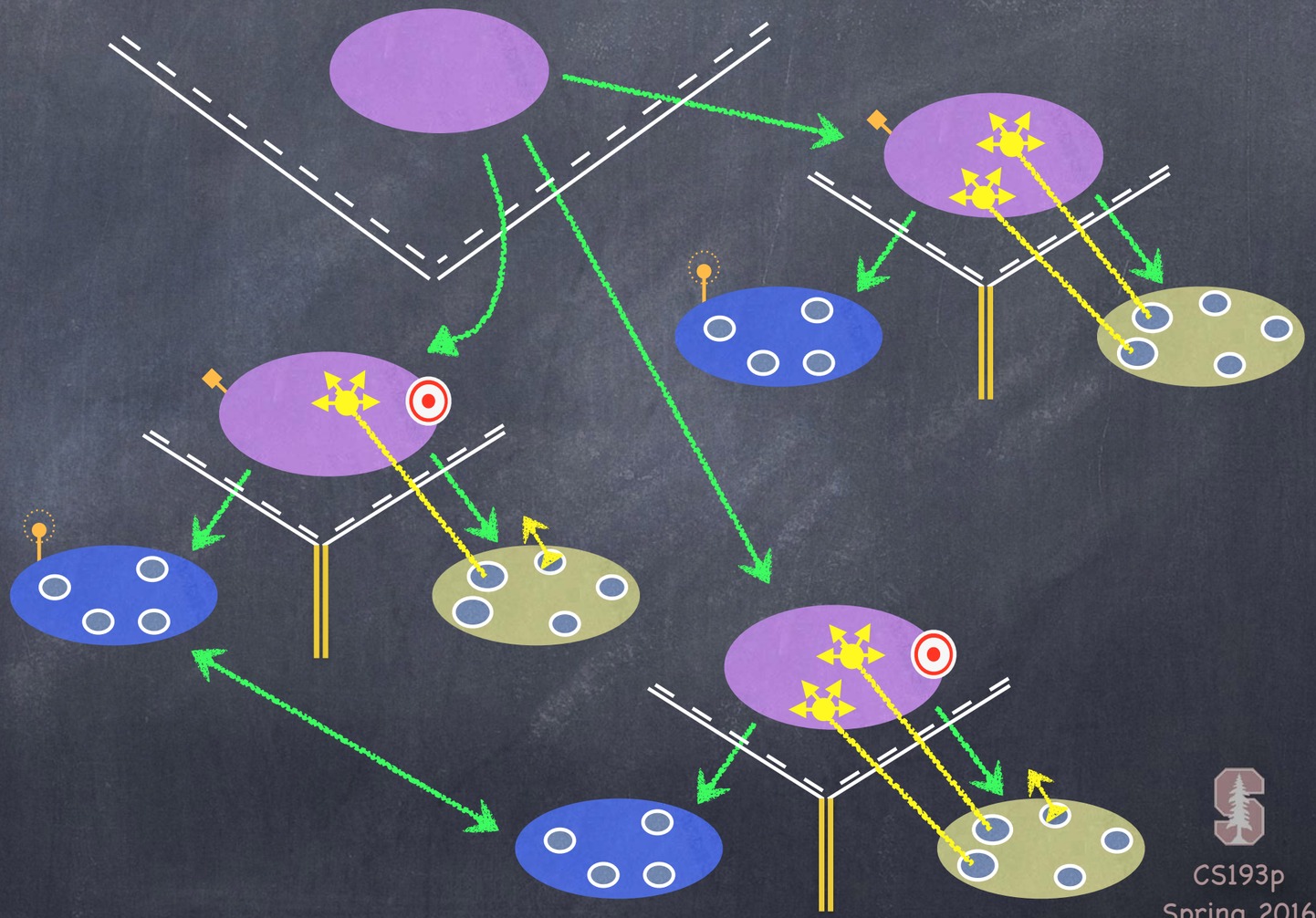
* you could build your own Controller that does this, but we're not going to cover that in this course

Multiple MVCs

- Time to build more powerful applications

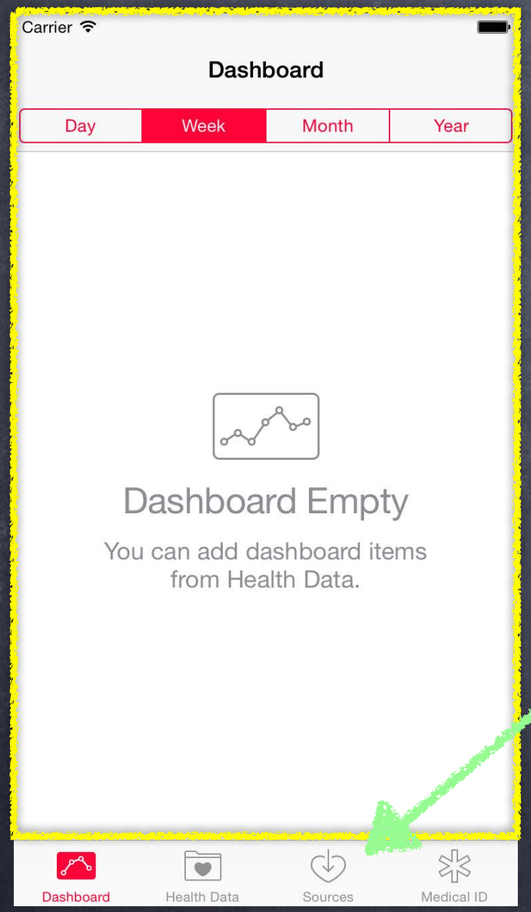
  To do this, we must combine MVCs …

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

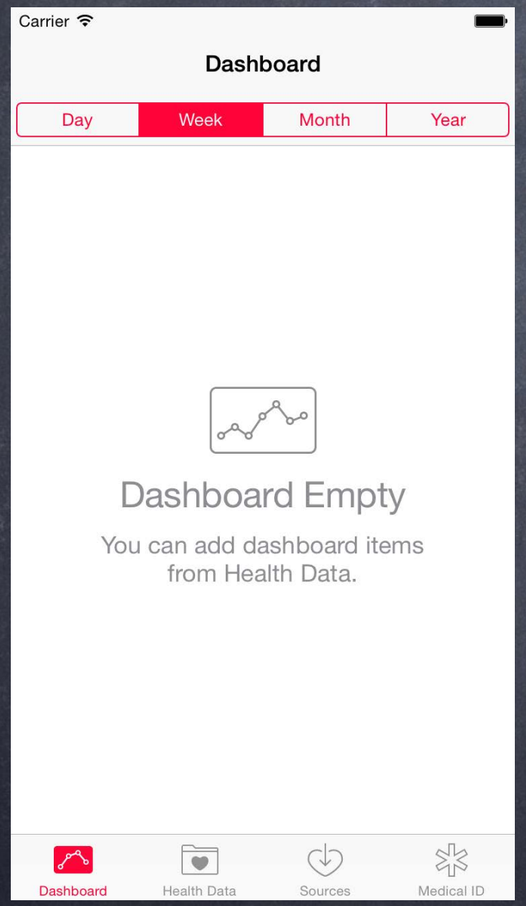
The icon, title and even a "badge value" on these is determined by the MVCs themselves via their property:

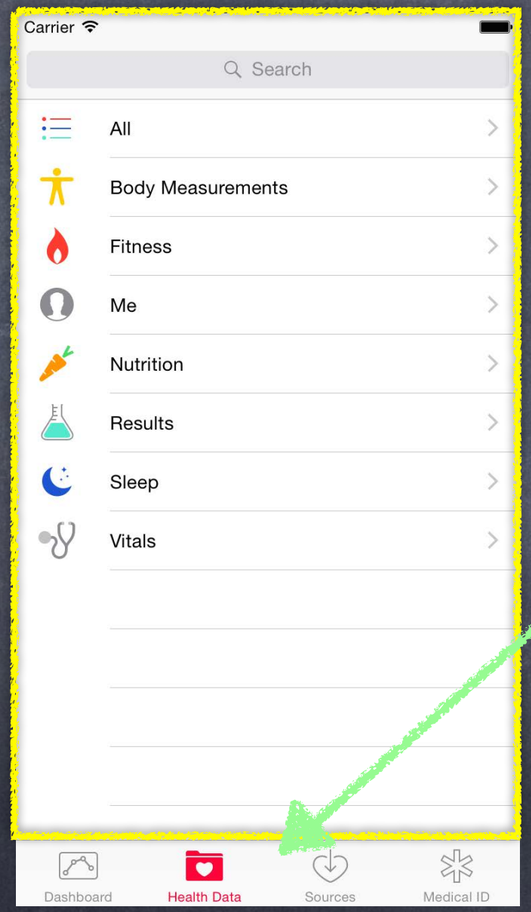
```
var tabBarItem: UITabBarItem!
```

But usually you just set them in your storyboard.

UITabBarController

- It lets the user choose between different MVCs …



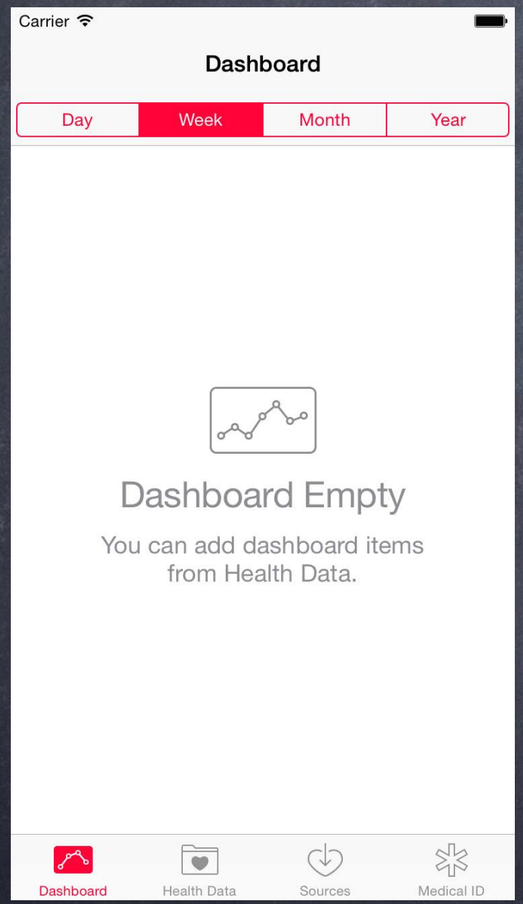


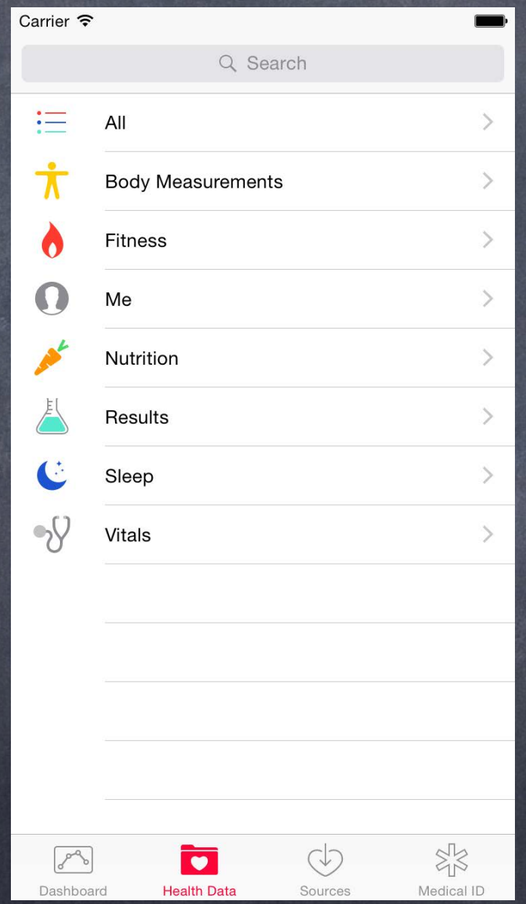
A "Health Data" MVC

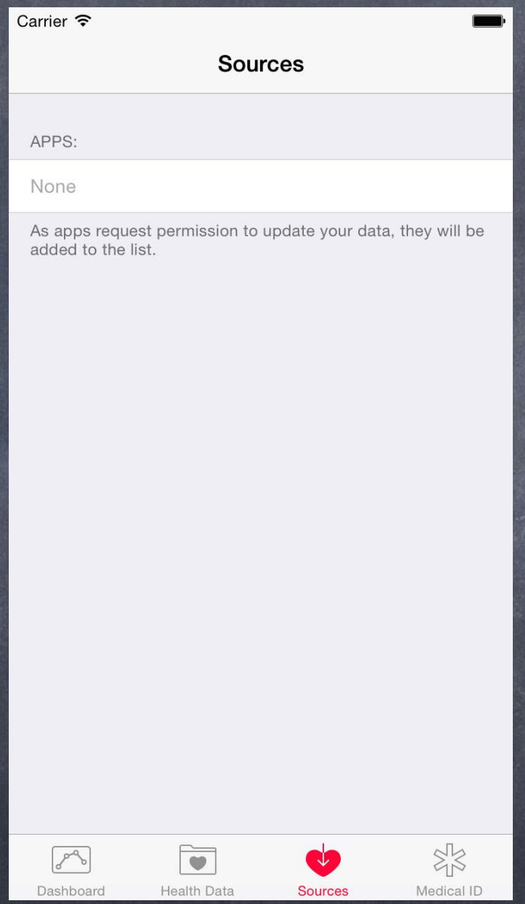
If there are too many tabs to fit here, the `UITabBarController` will automatically present a UI for the user to manage the overflow!

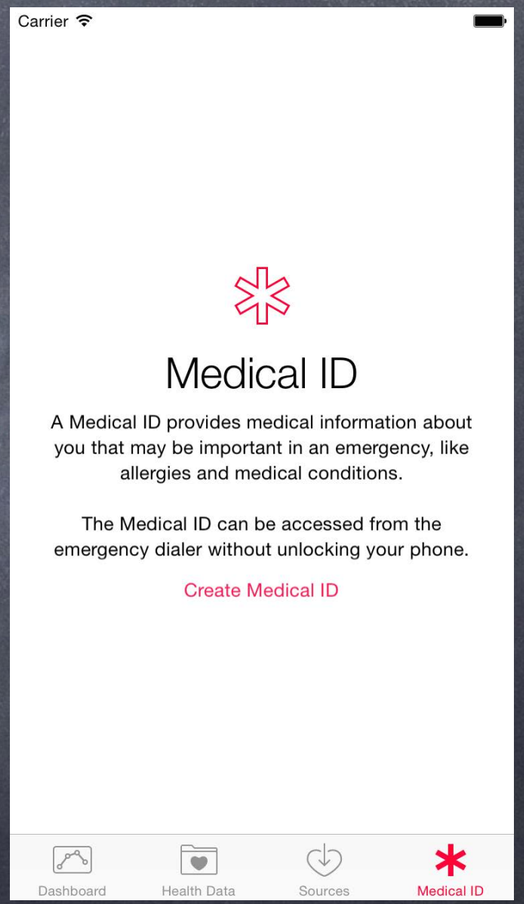
UITabBarController

- It lets the user choose between different MVCs …







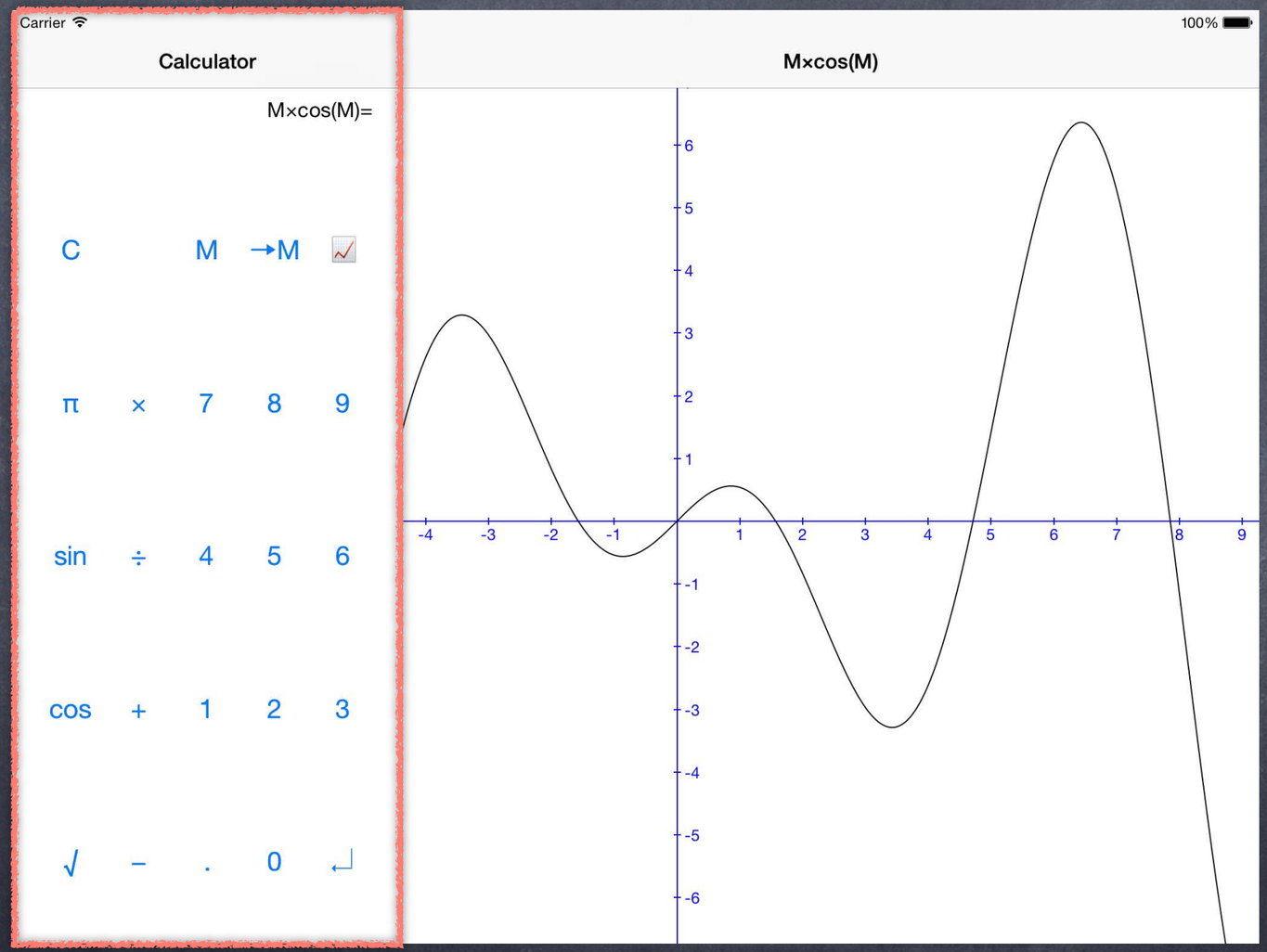


UISplitViewController

- Puts two MVCs side-by-side …

A Calculator MVC

Master

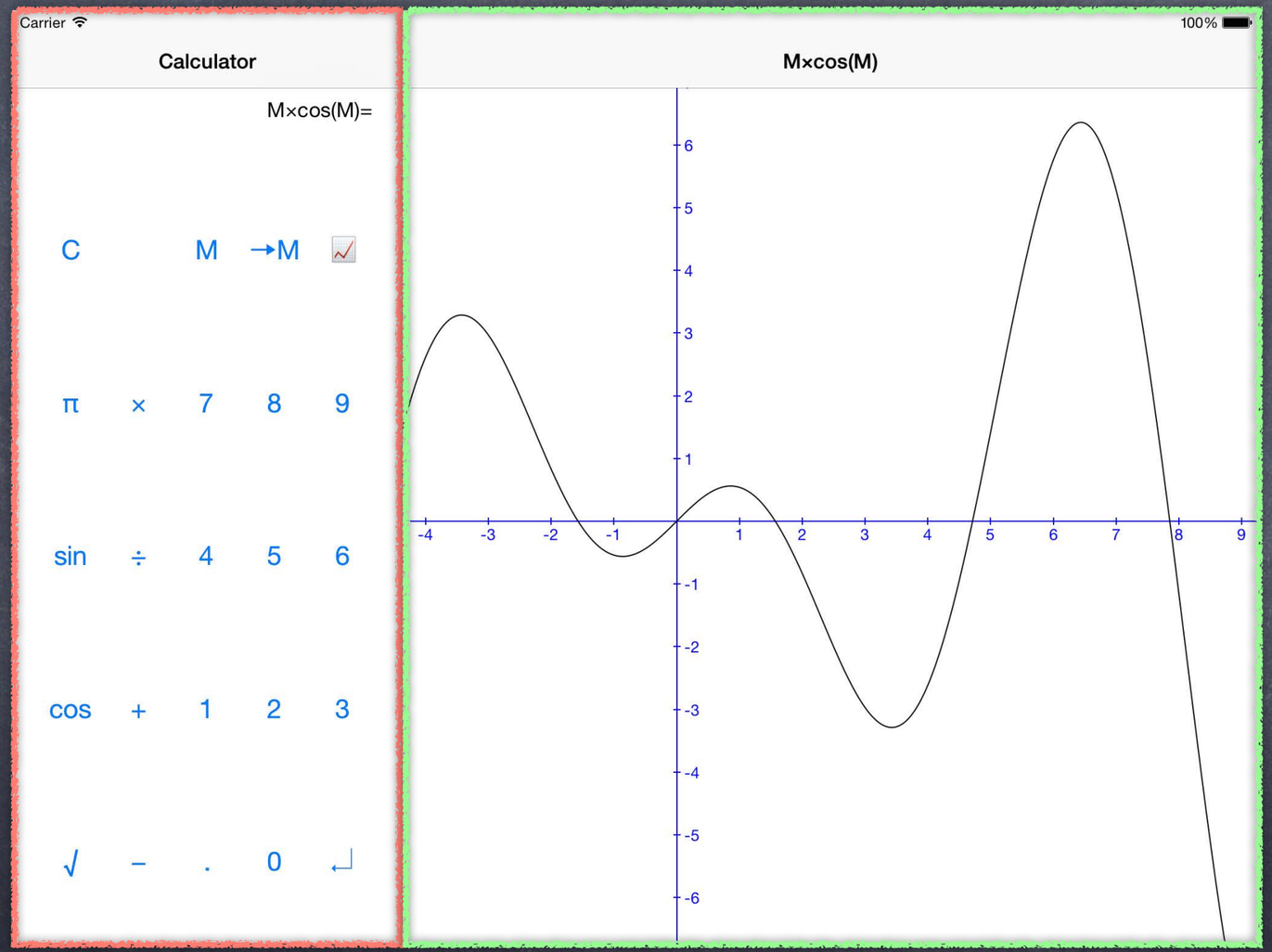


CS193p
Spring 2016

UISplitViewController

- Puts two MVCs side-by-side …

A Calculator MVC



A Calculator Graph MVC

Master

Detail

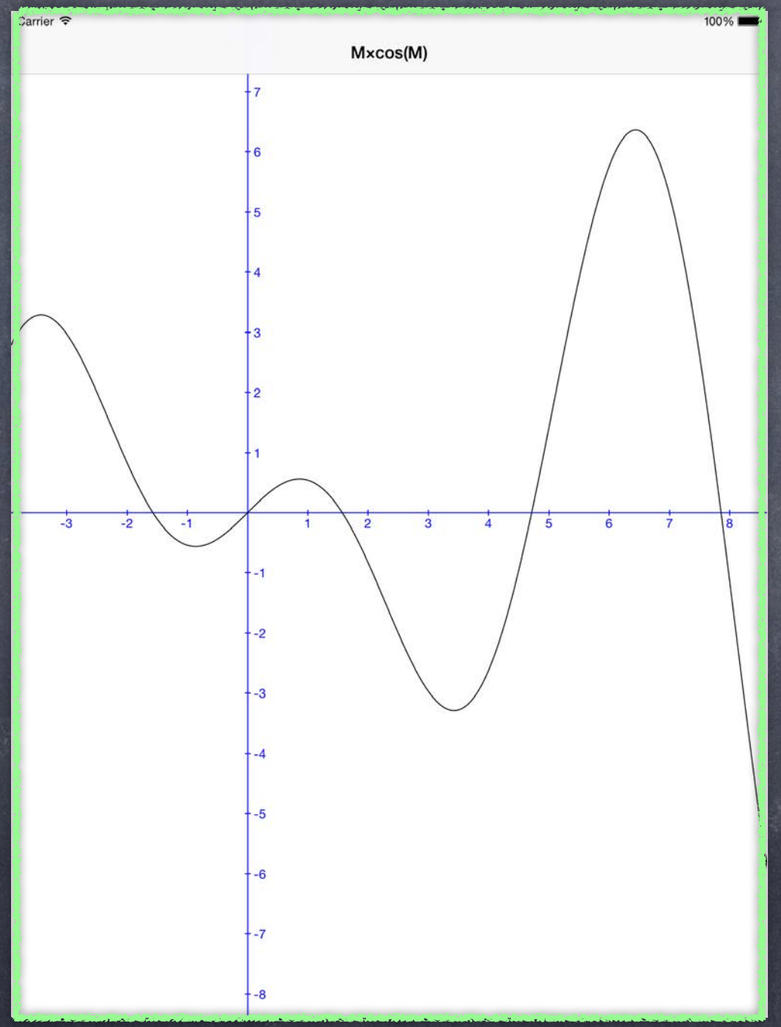
CS193p
Spring 2016

UISplitViewController

- Puts two MVCs side-by-side …

A
Calculator
MVC

Master



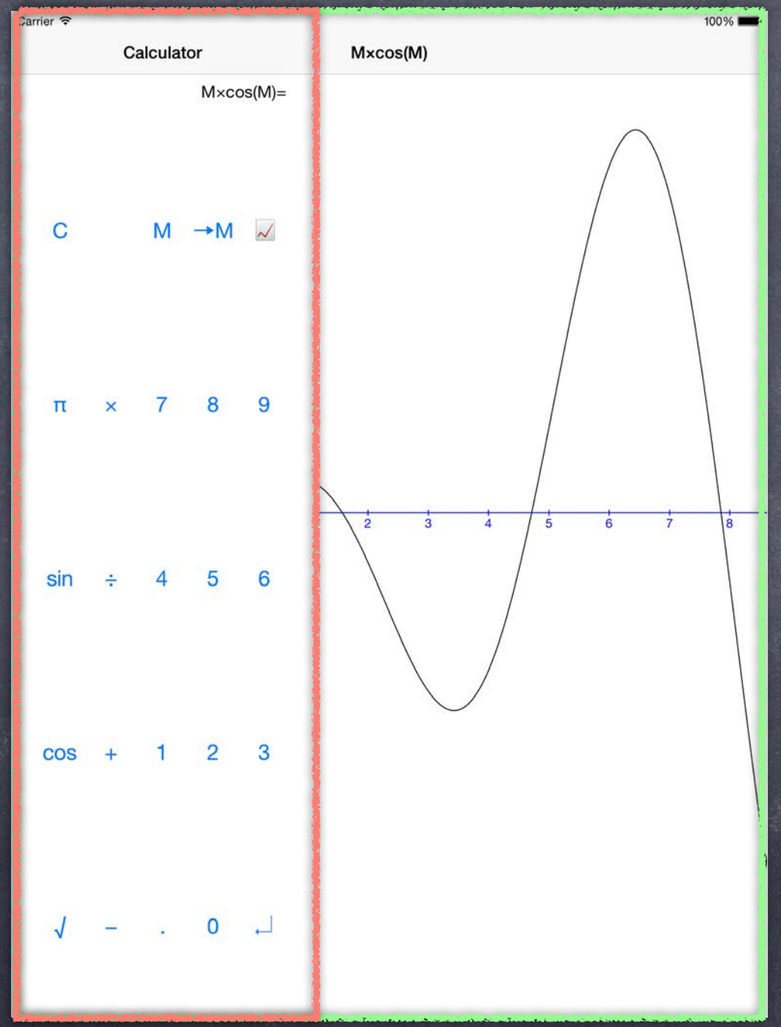
A
Calculator Graph
MVC

Detail

UISplitViewController

- Puts two MVCs side-by-side …

A
Calculator
MVC



A
Calculator Graph
MVC

Master

Detail

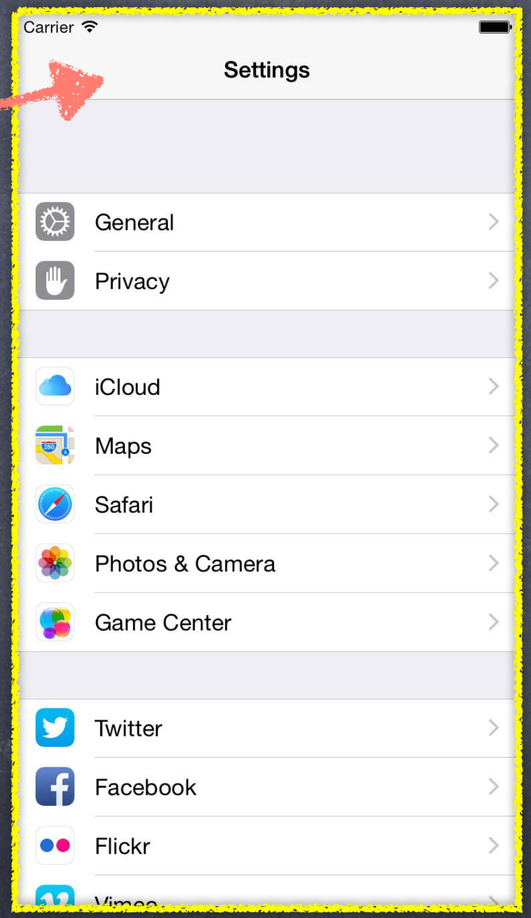
UINavigationController

- 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)

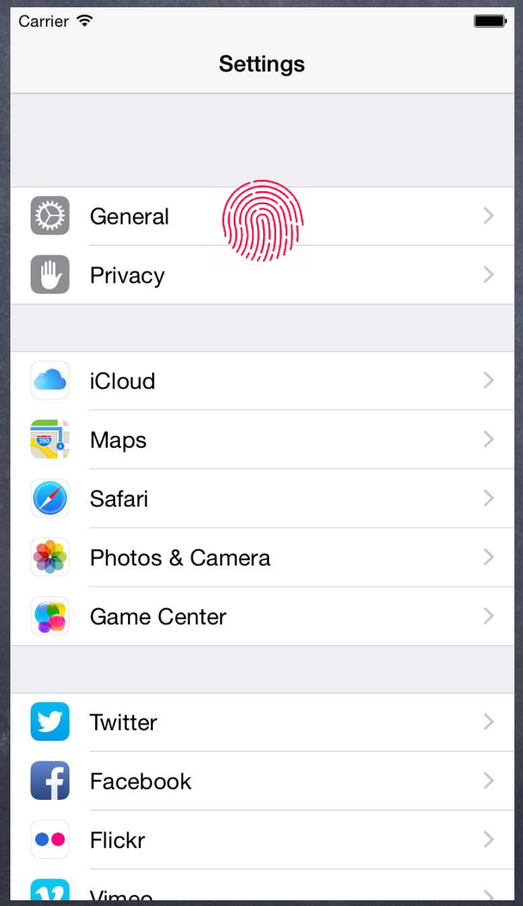
Each MVC communicates these contents via its `UIViewController`'s `navigationItem` property



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

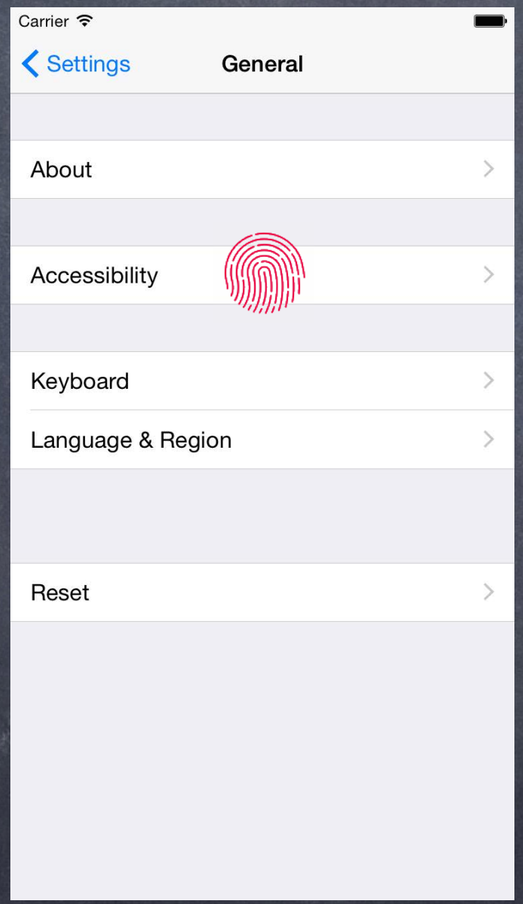
- Pushes and pops MVCs off of a stack (like a stack of cards) …

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) …



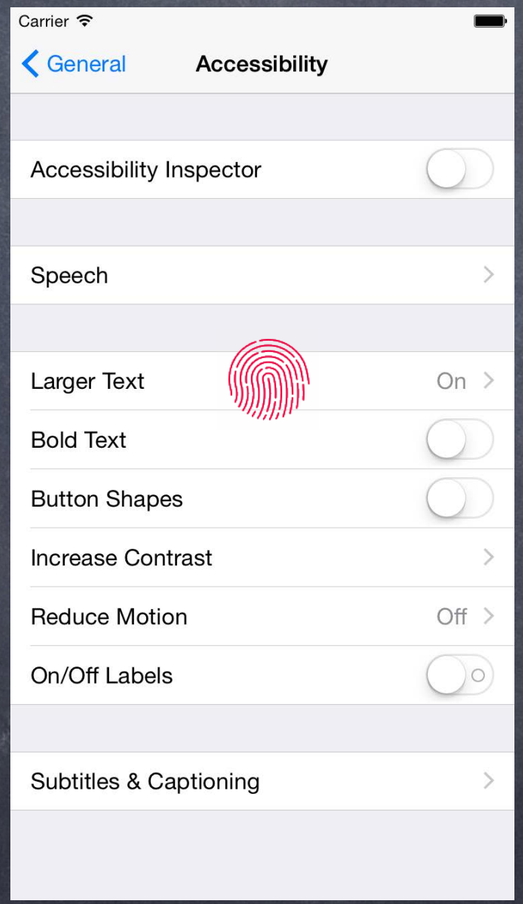
UINavigationController

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

An "Accessibility" MVC

UINavigationController

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



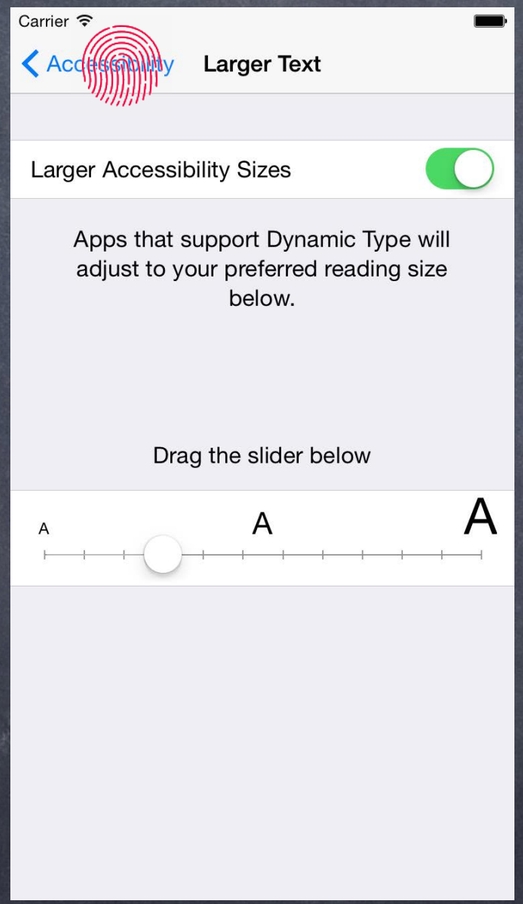
UINavigationController

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

A "Larger Text" MVC

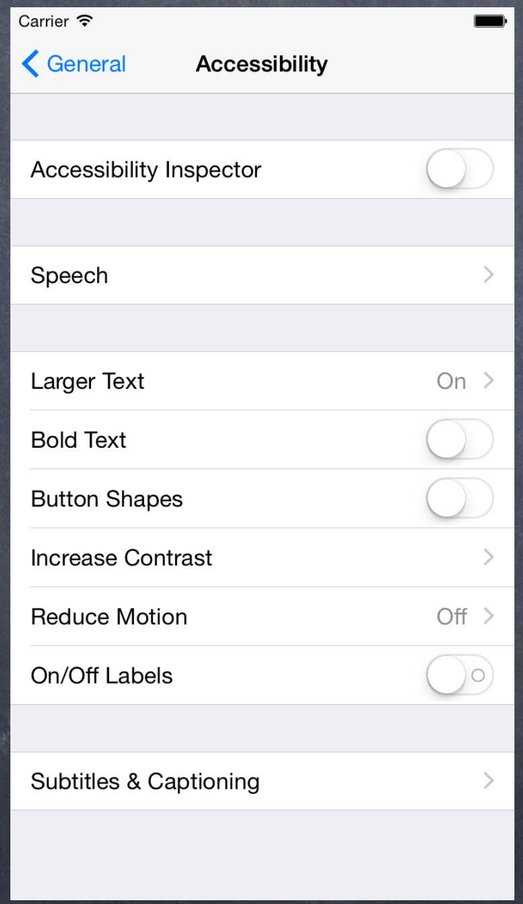
UINavigationController

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



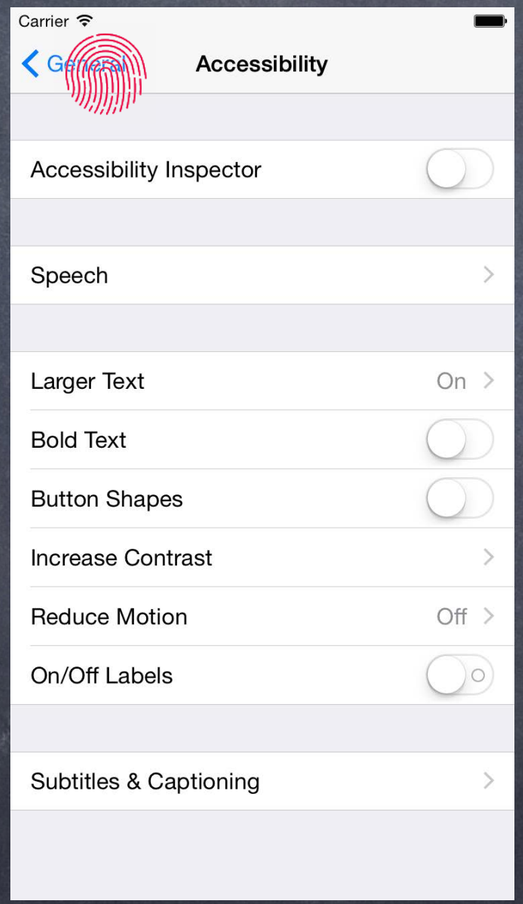
UINavigationController

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



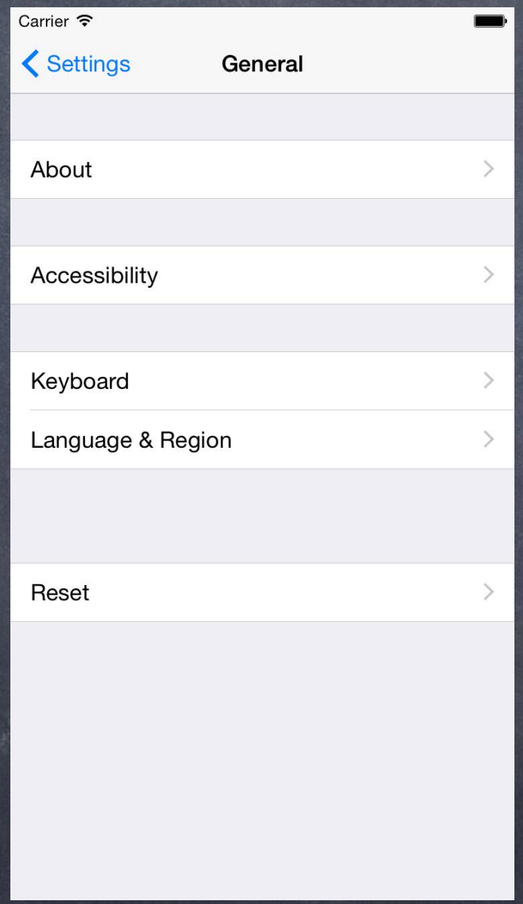
UINavigationController

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



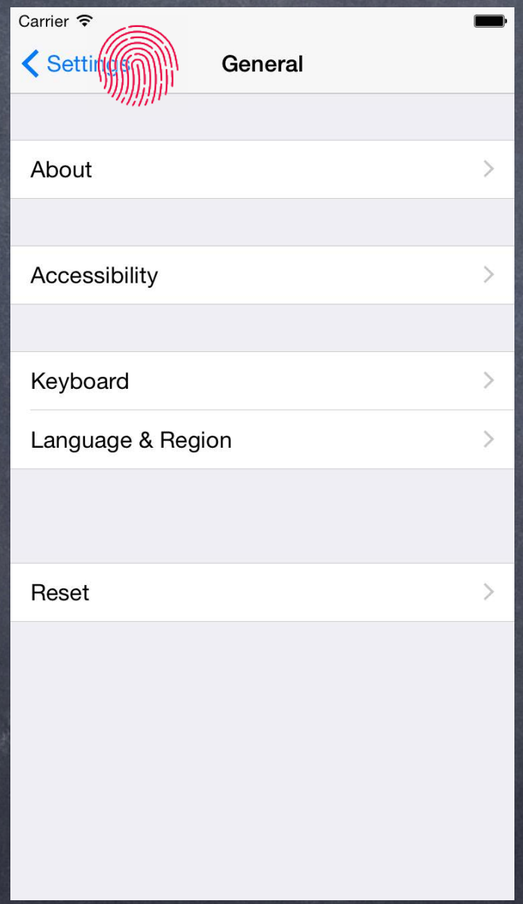
UINavigationController

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



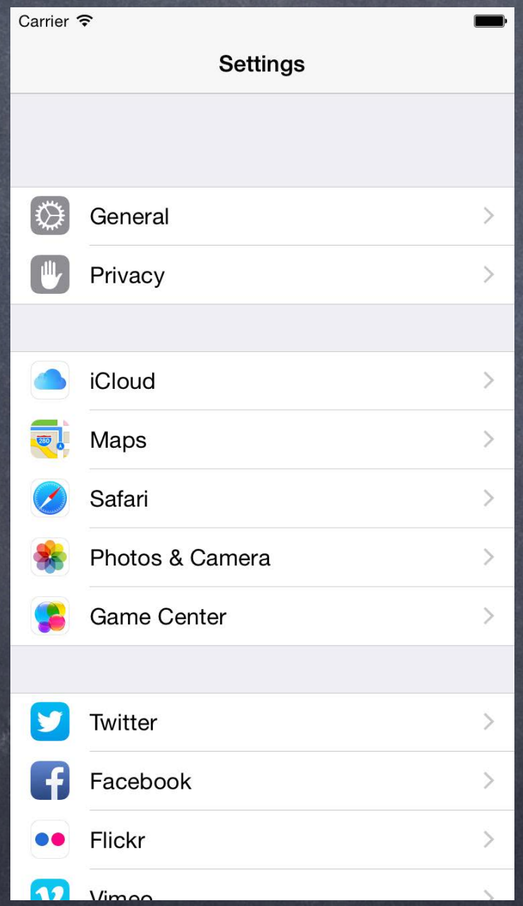
UINavigationController

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

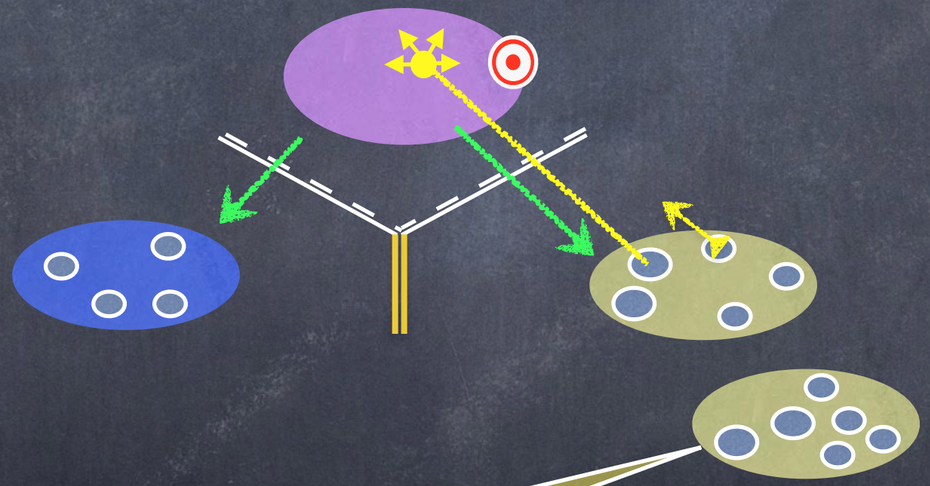


UINavigationController

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

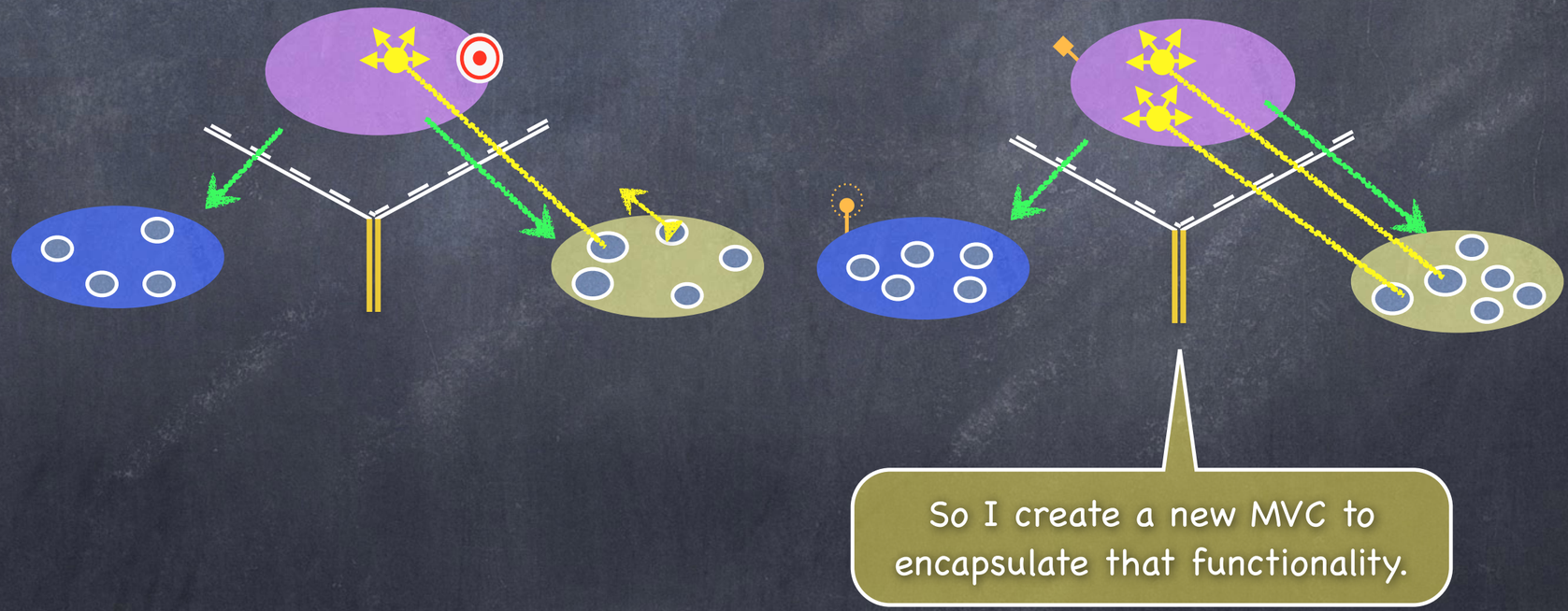


UINavigationController



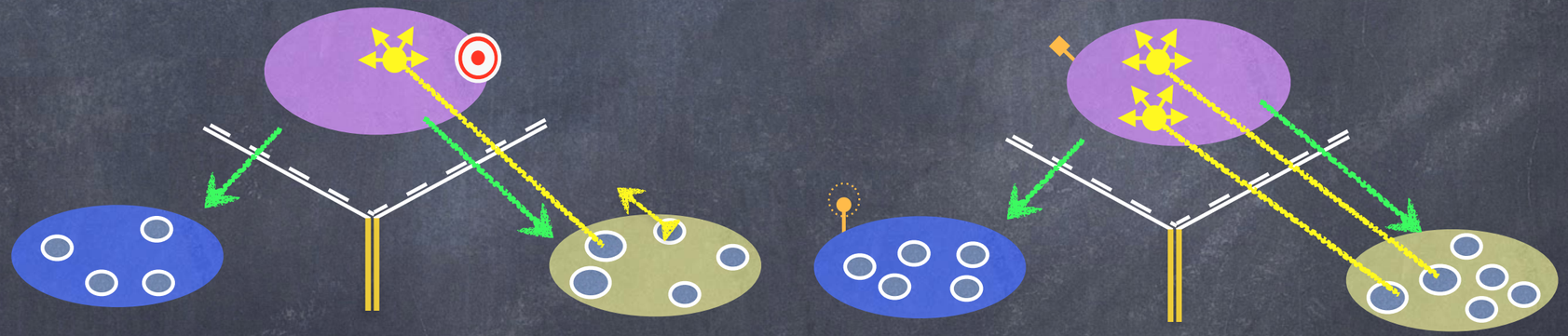
I want more features, but it doesn't make sense to put them all in one MVC!

UINavigationController

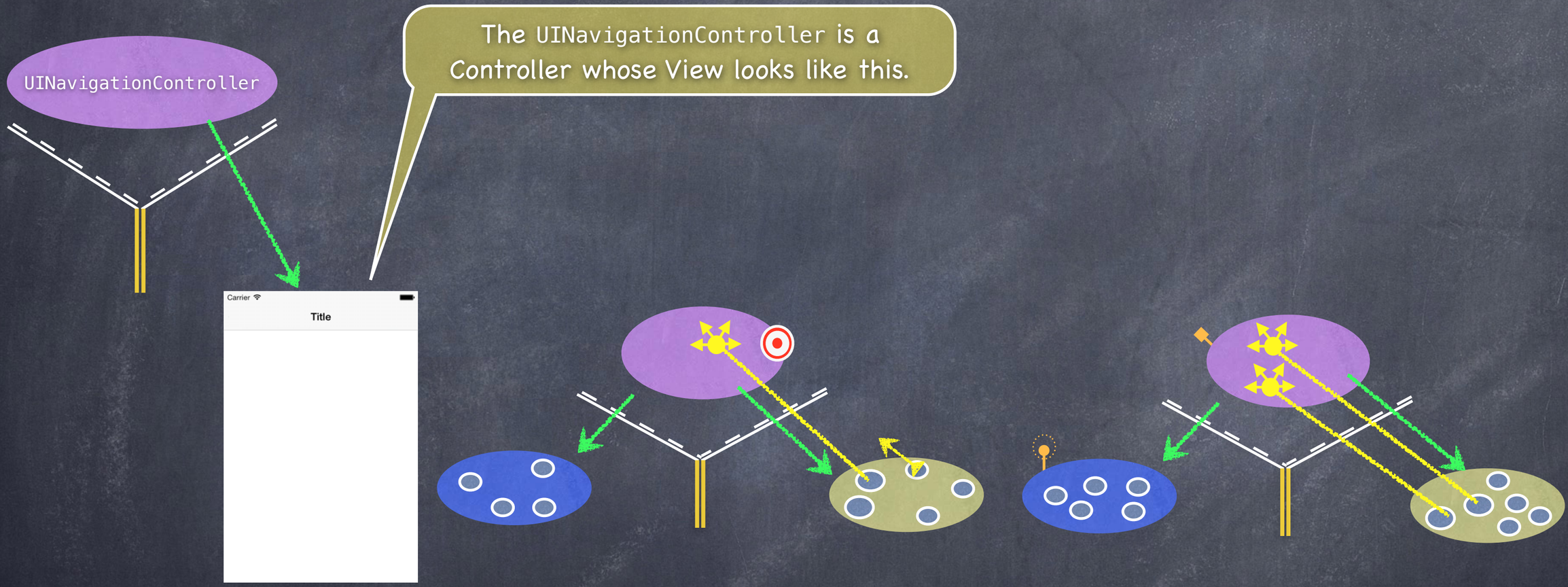


UINavigationController

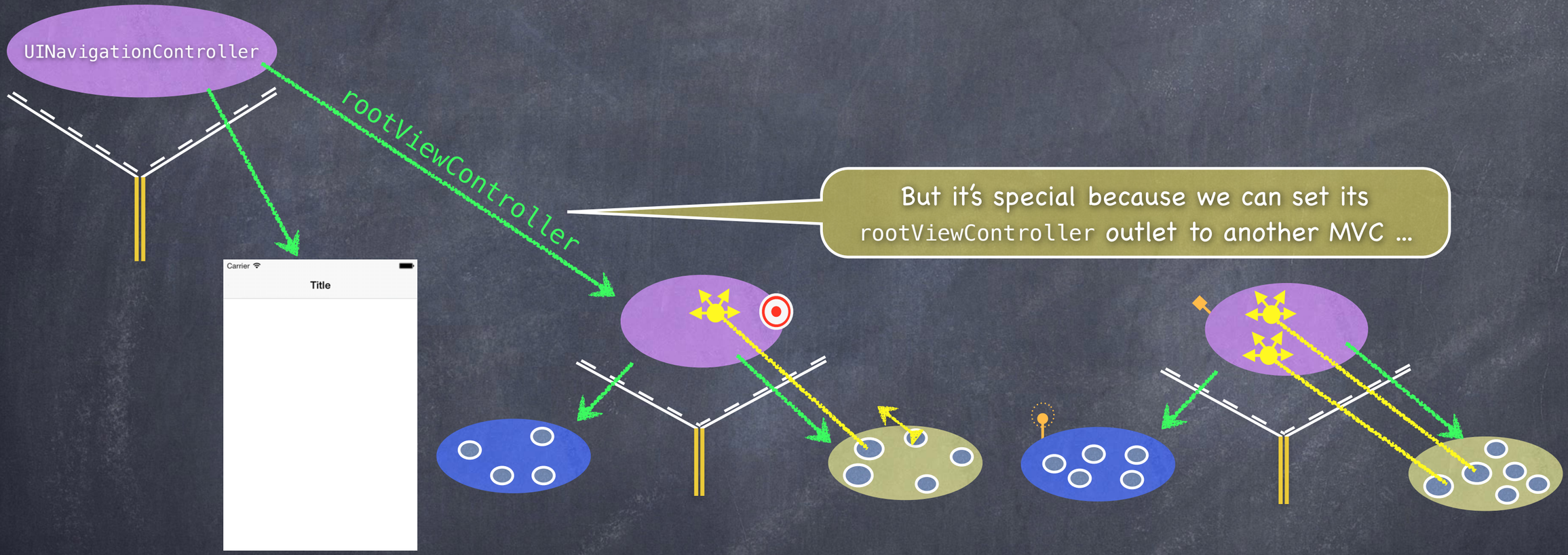
We can use a UINavigationController to let them share the screen.



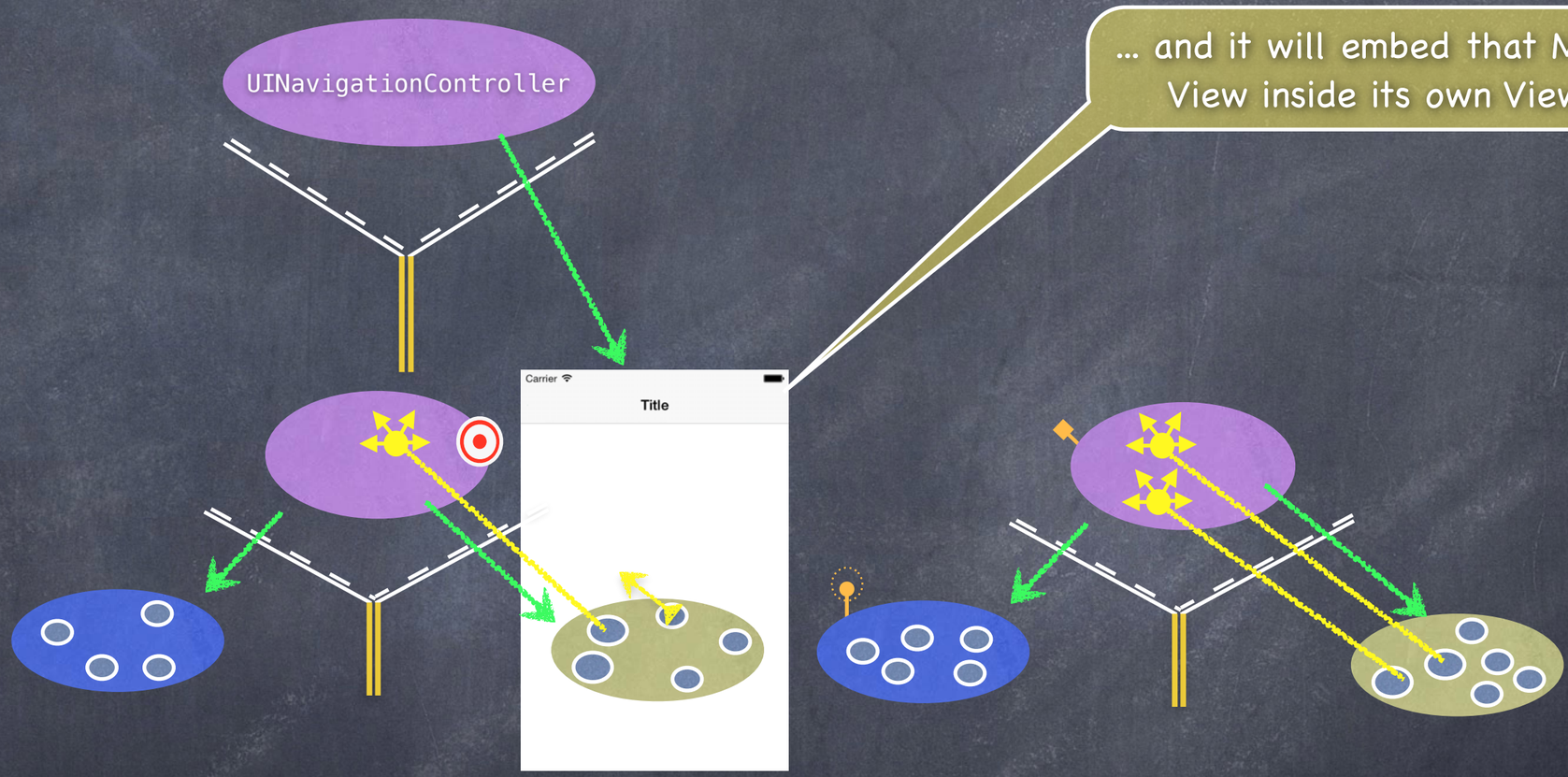
UINavigationController



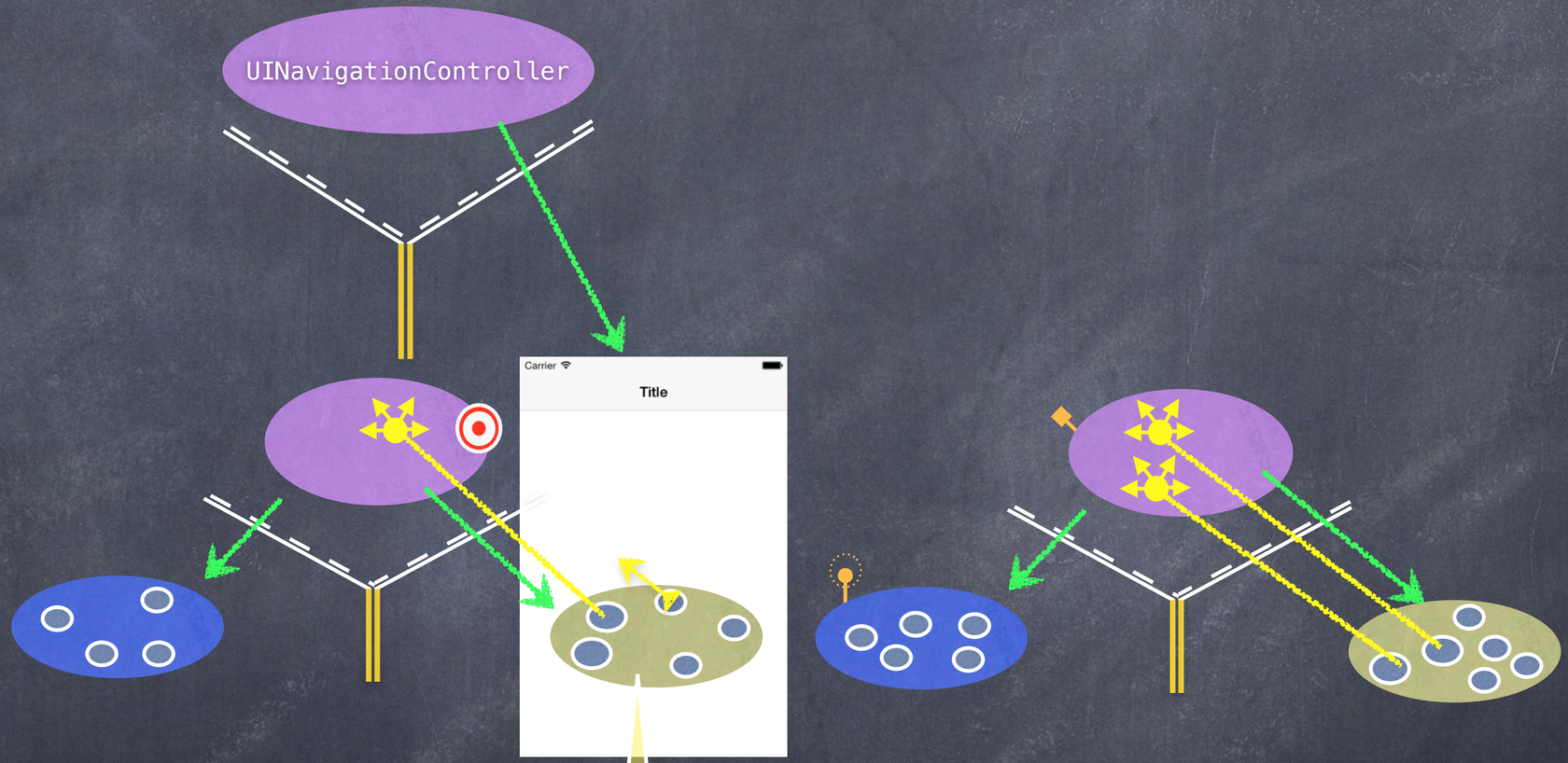
UINavigationController



UINavigationController

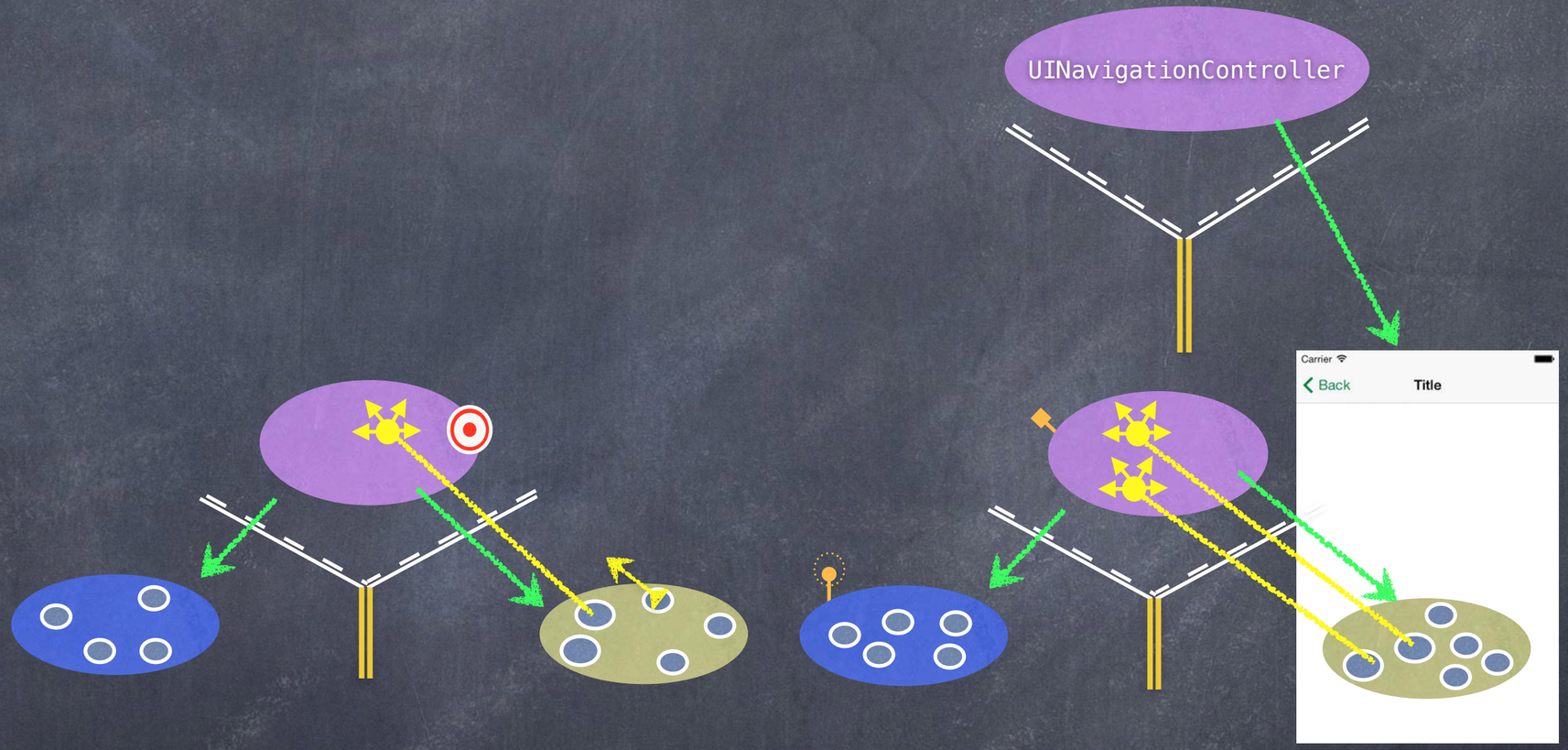


UINavigationController



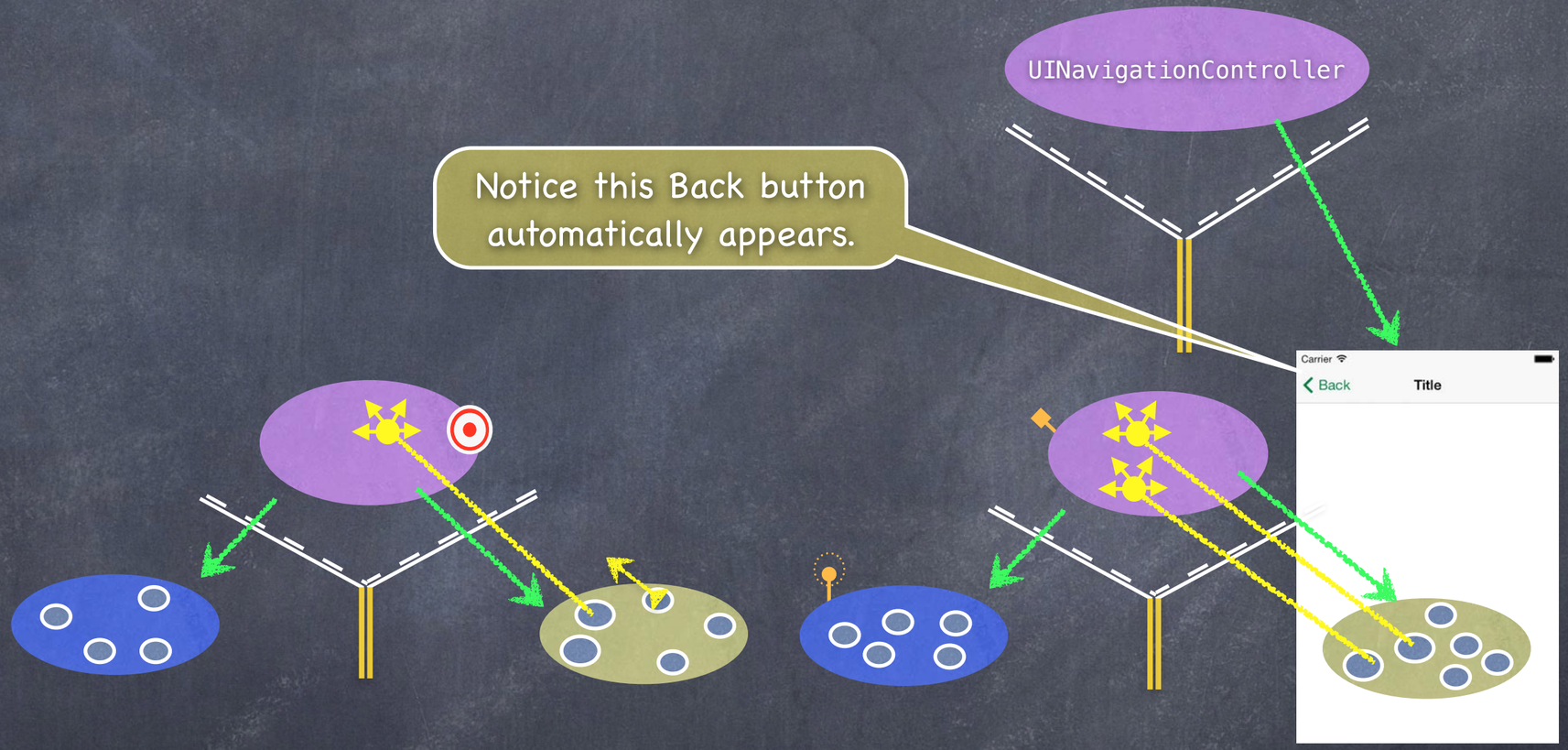
Then a UI element in this View (e.g. a `UIButton`) can segue to the other MVC and its View will now appear in the `UINavigationController` instead.

UINavigationController

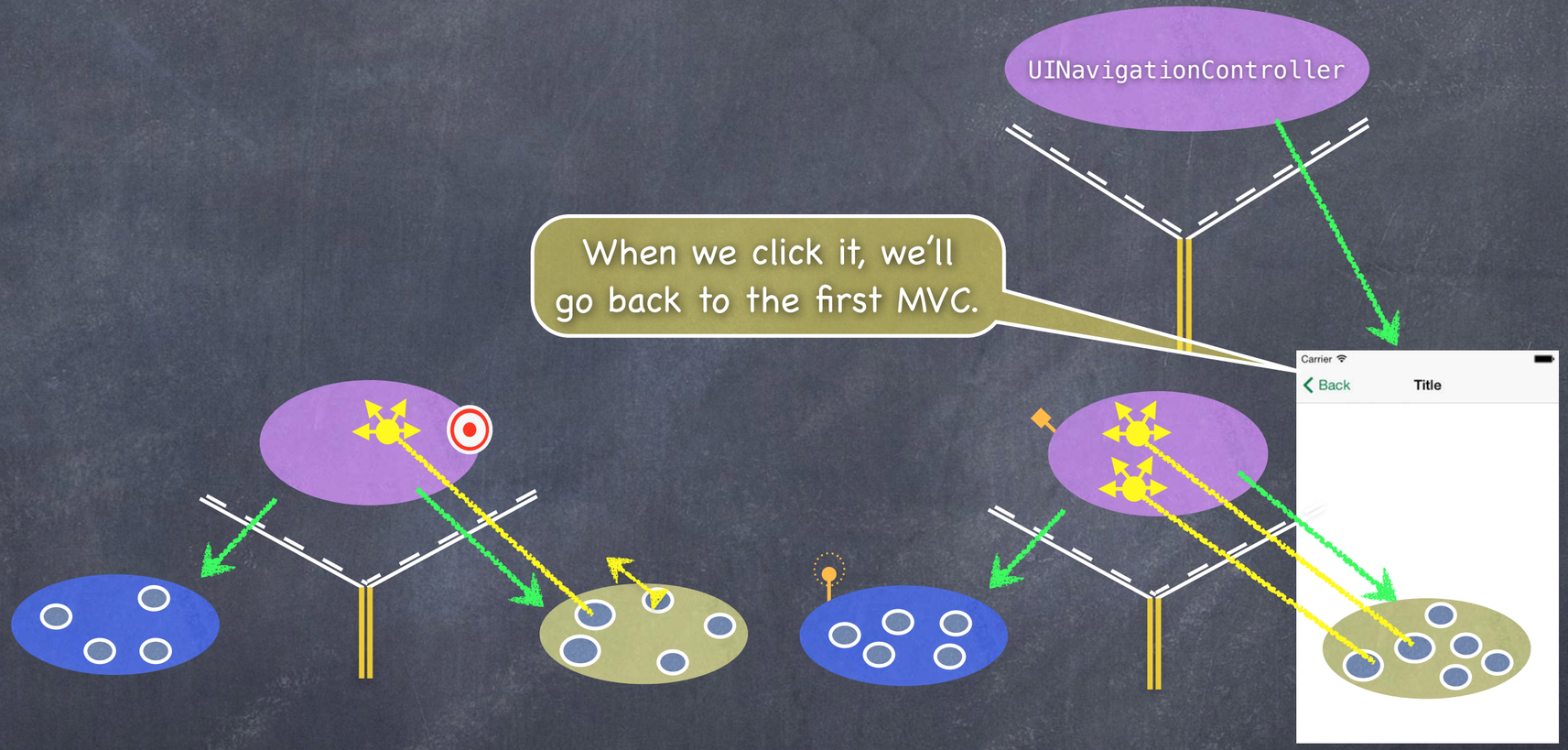


We call this kind of segue a "Show (push) segue".

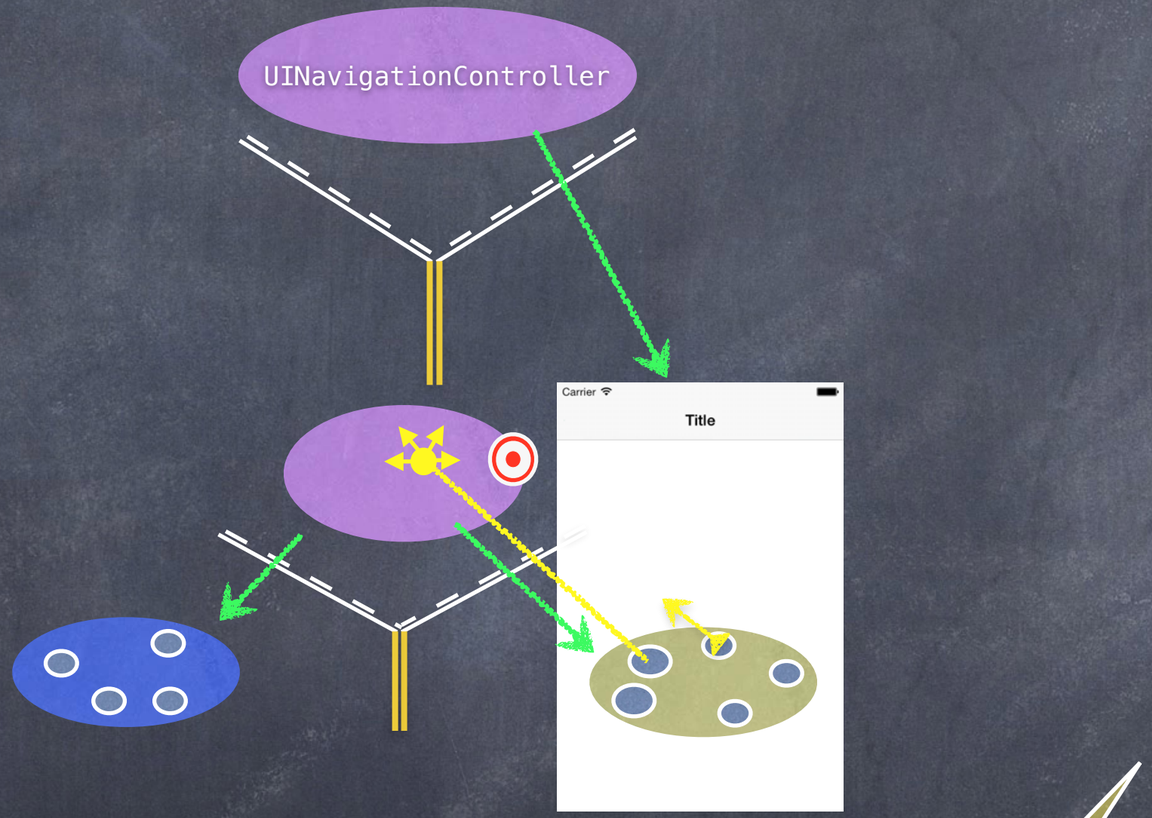
UINavigationController



UINavigationController

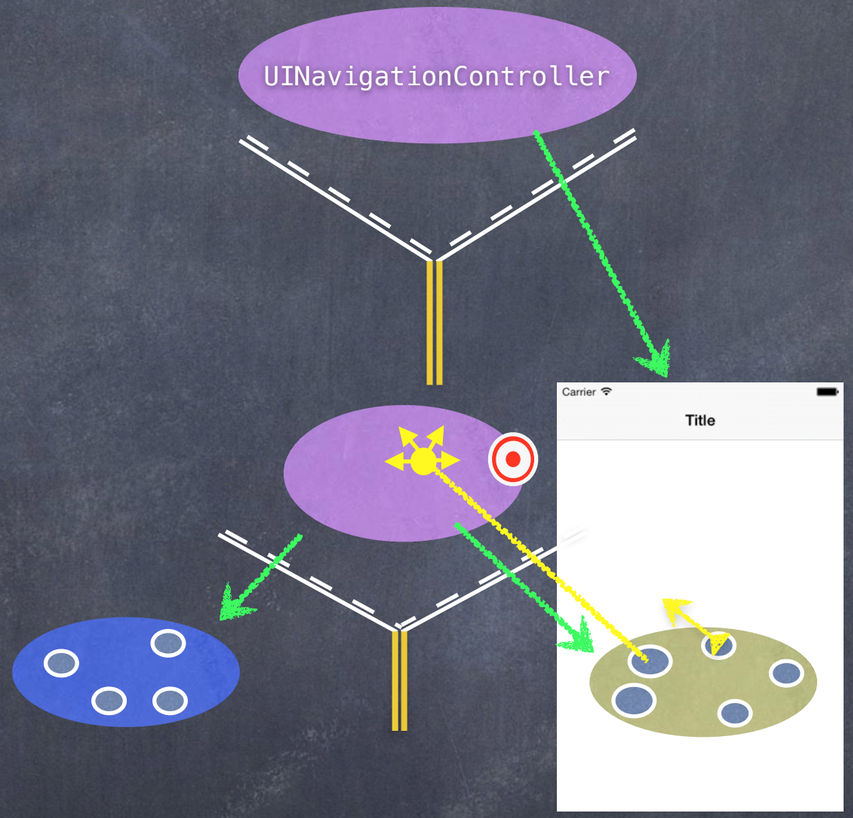


UINavigationController



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

UINavigationController



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 of the split view controller you are in …

```
if let detailVC: 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

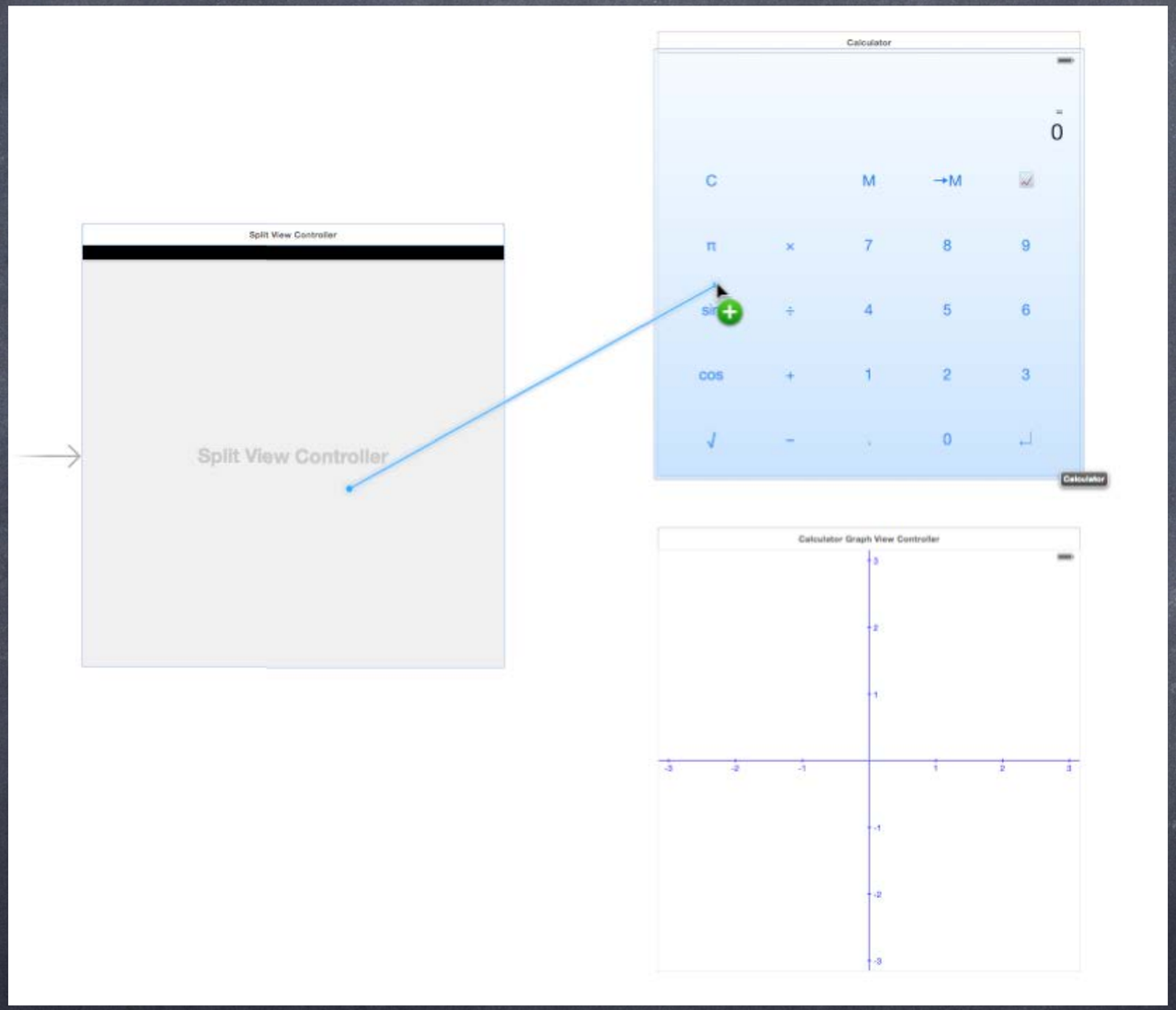
  How do we hook them up to be the two sides of a Split View?

  Just drag out a 

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

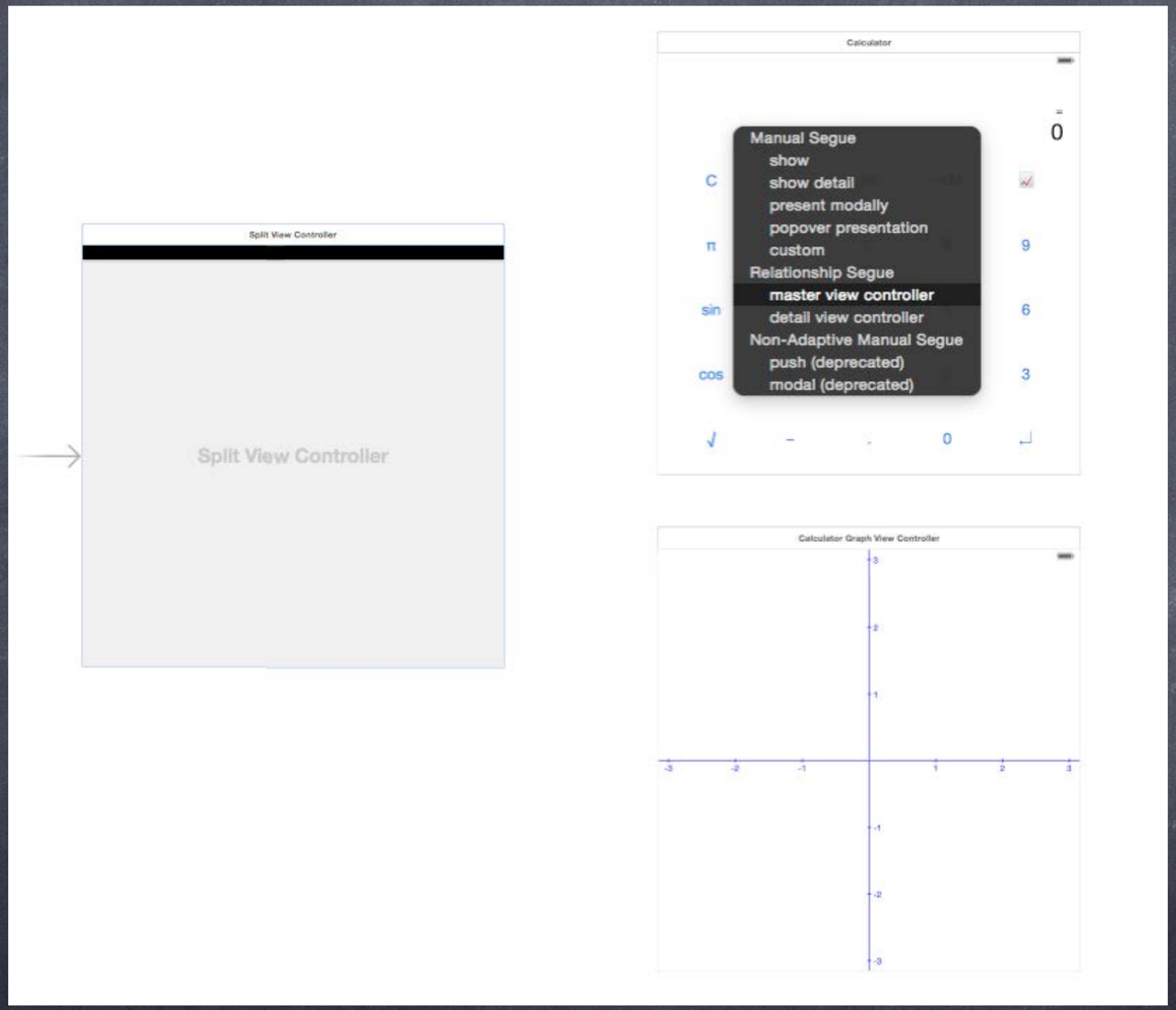
  Then ctrl-drag from the `UISplitViewController` to the master and detail MVCs ...

Wiring up MVCs

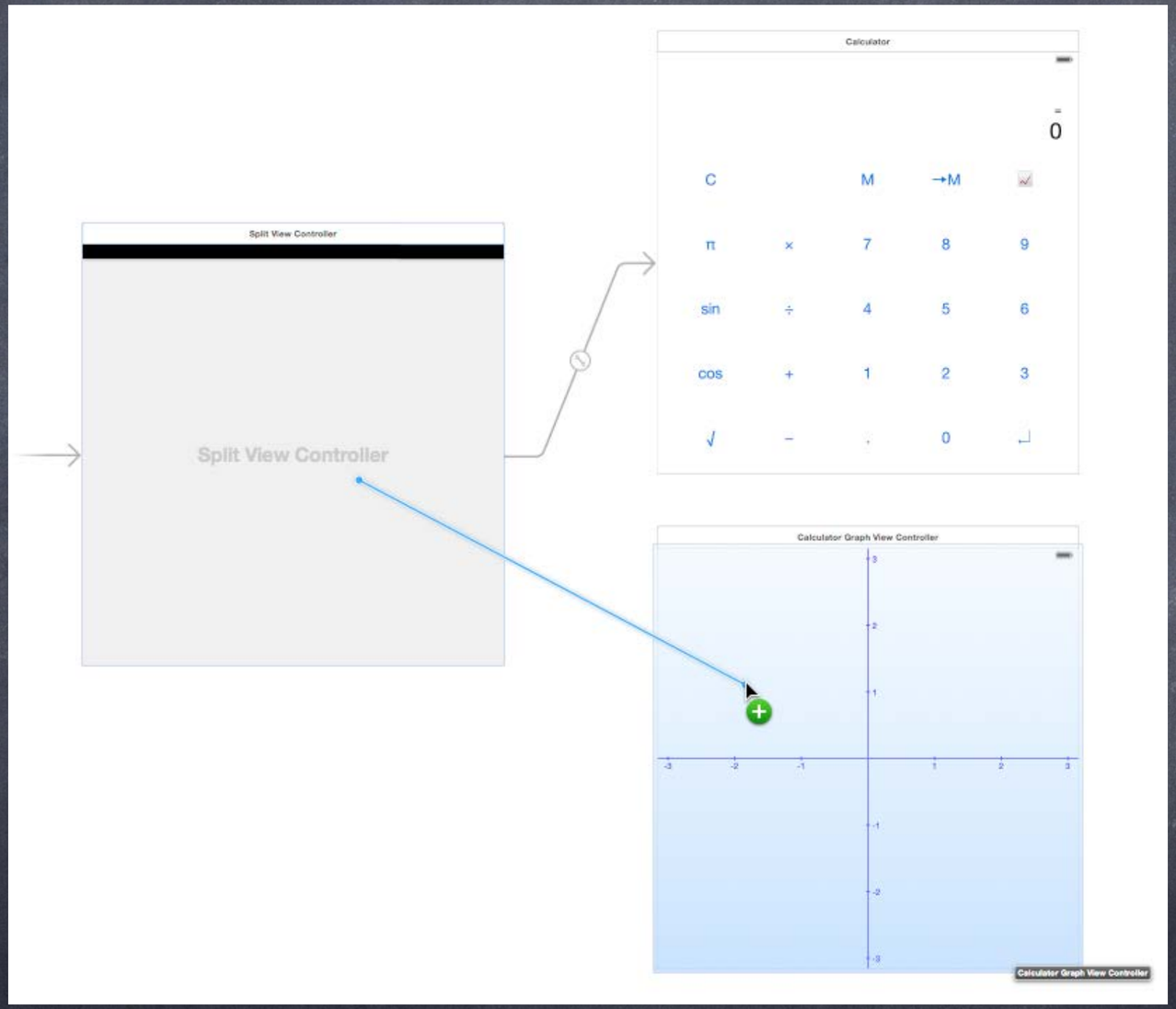


CS193p
Spring 2016

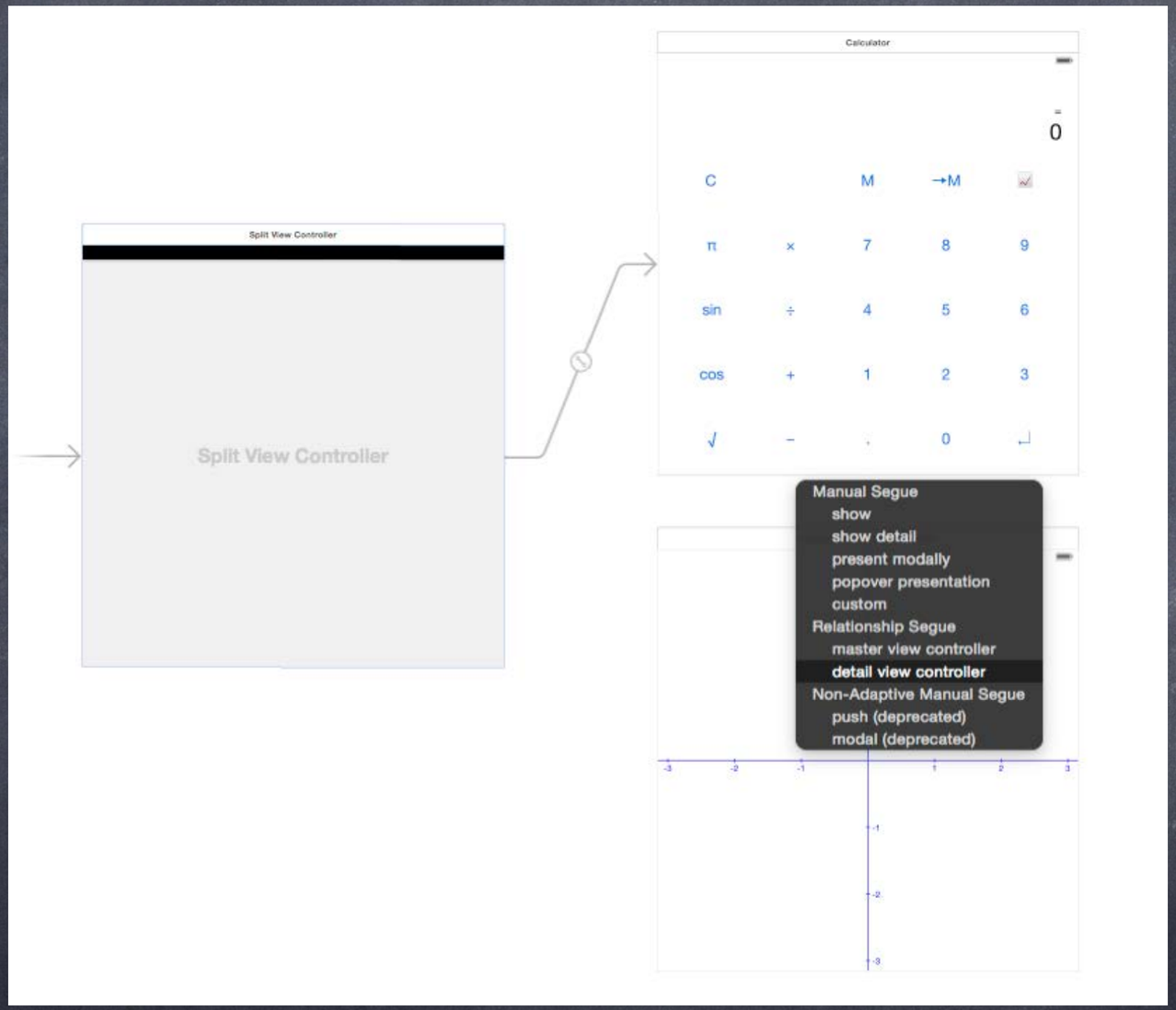
Wiring up MVCs



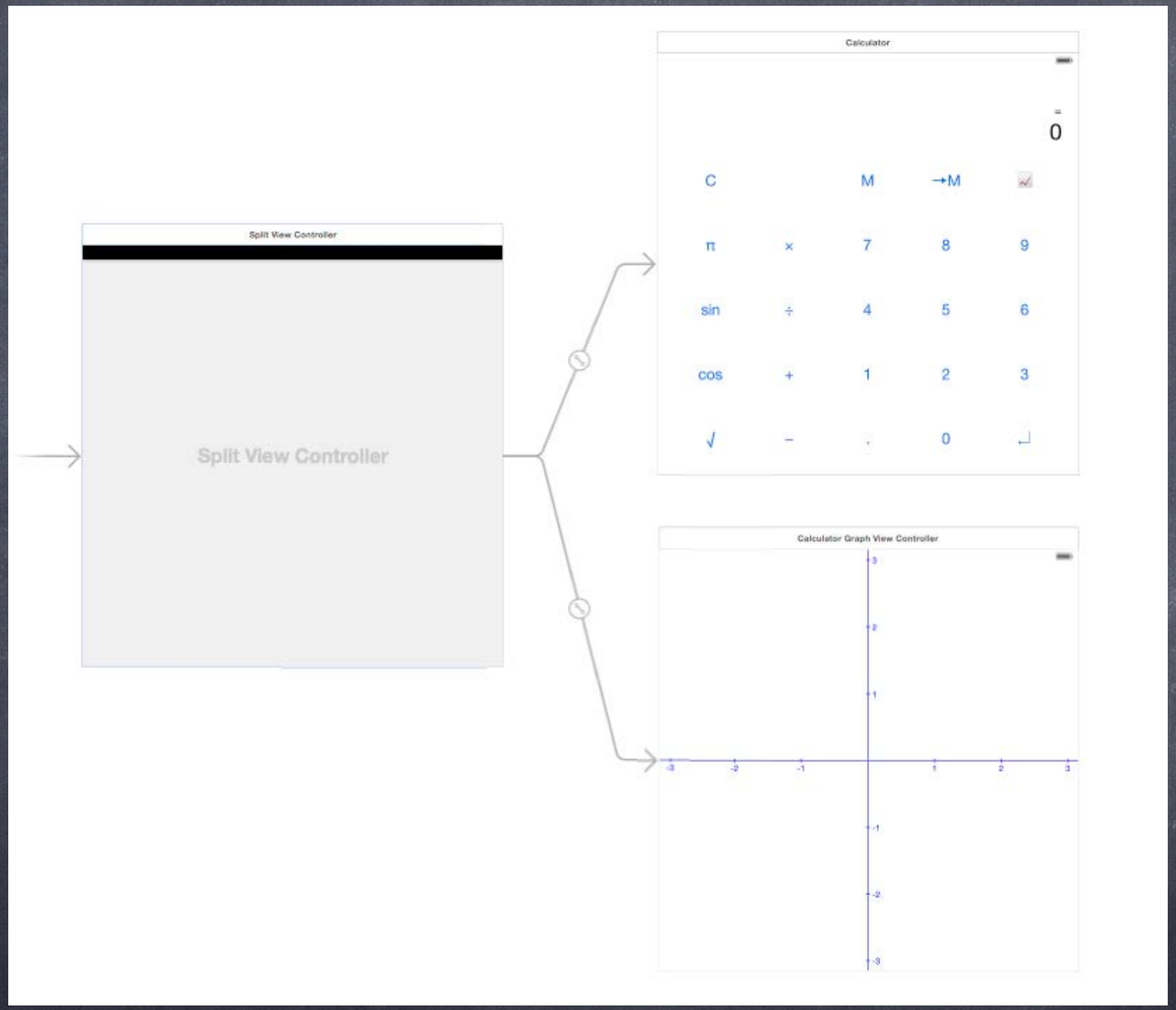
Wiring up MVCs



Wiring up MVCs

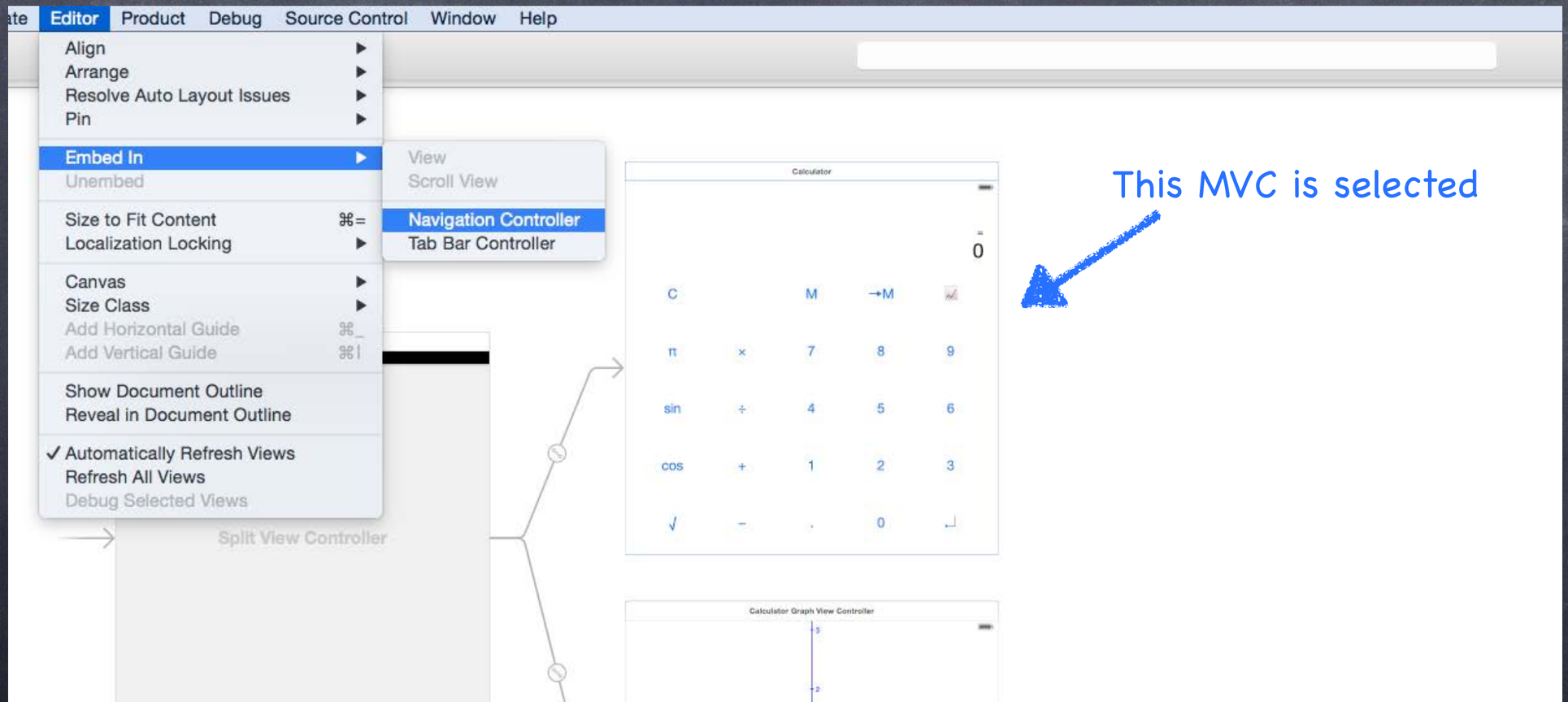


Wiring up MVCs



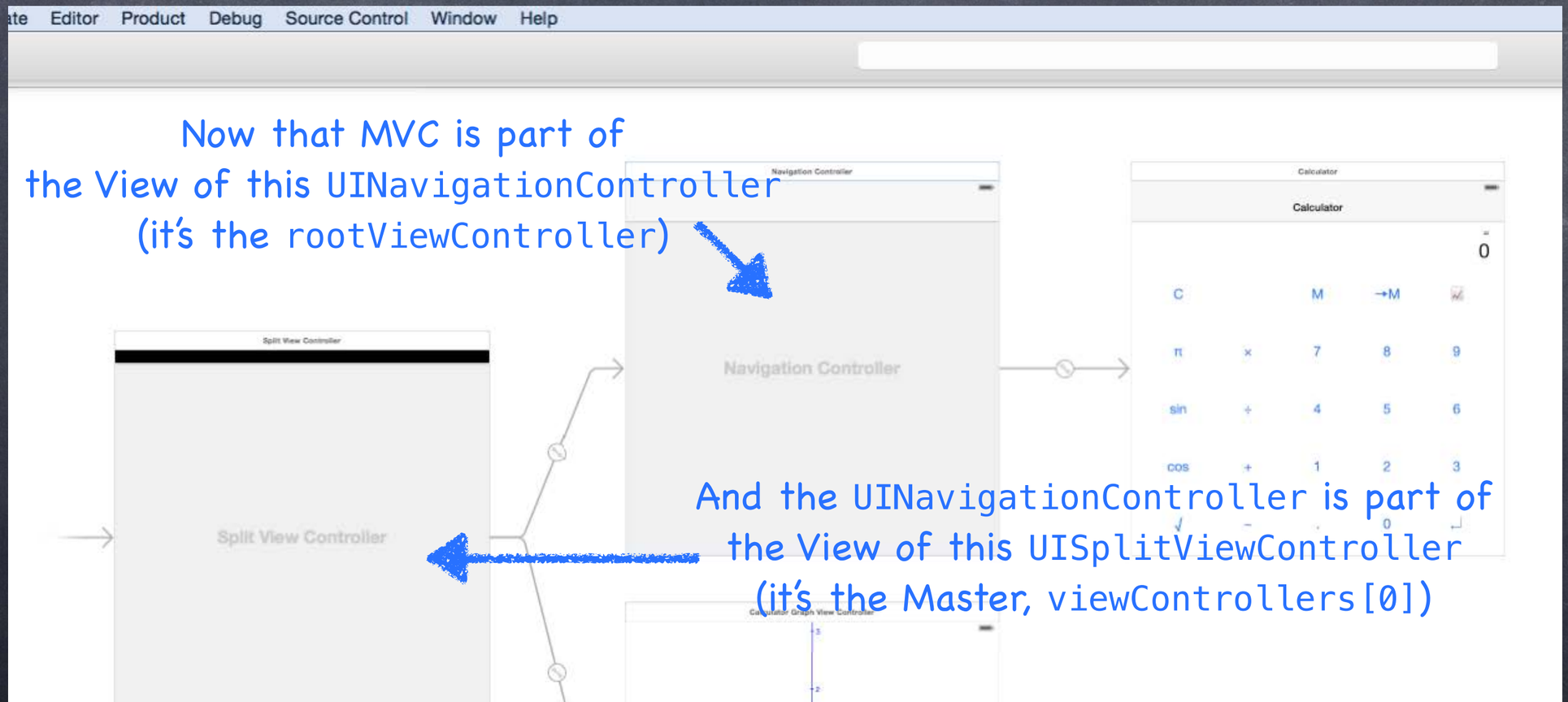
Wiring up MVCs

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



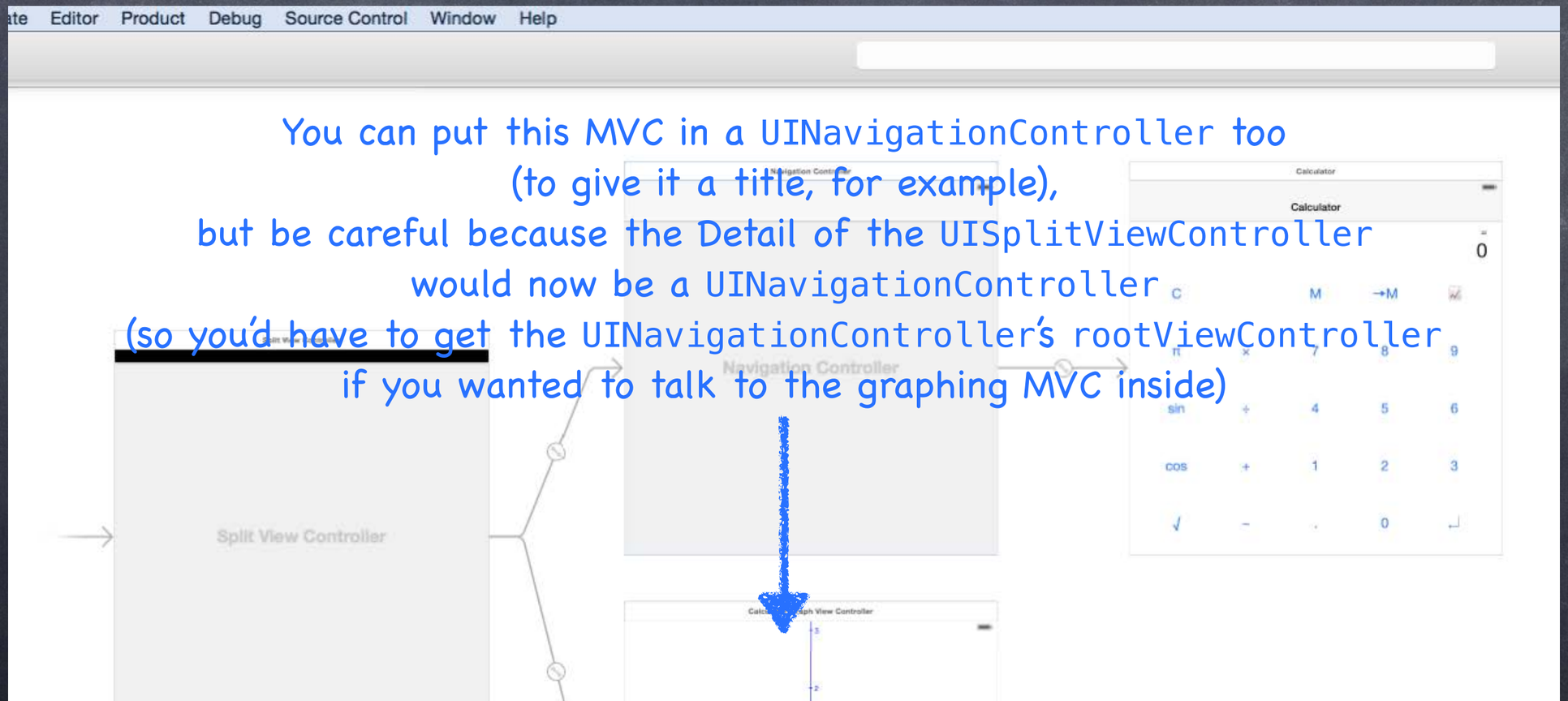
Wiring up MVCs

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



Wiring up MVCs

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



Wiring up MVCs

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