Desarrollo de Apps para iOS
View Controllers

IWEB,LSWC 2014-2015
Santiago Pavón

ver: 2015-04-08

Una Pantalla

- Quiero hacer una clase que represente una pantalla de mi aplicación.
  - Toda la pantalla en un iPhone, parte de la de iPad,…
- Será una clase controladora (MVC) que:
  - mostrará una vista (formada por una jerarquía de views) con los datos del modelo.
  - actualizará el modelo según se interactúe con la vista.
- Que avise cuando haya problemas de memoria.
  - Para liberar los objetos que no necesite.
    - Más tarde los reconstruiré otra vez, cuando los vuelva a necesitar.
- Que me avise cuando la pantalla vaya a hacerse visible, o se haya hecho visible.
  - Y al revés: cuando vaya a dejar de ser visible, o ya no sea visible.
- Que me ayude a presentar correctamente mi vista cuando giro el terminal.
- Que me ayude en la gestión de la navegación:
  - presentar otras pantallas, poder volver a las pantallas anteriores, tener pestañas para elegir que pantalla veo, etc…
- y más cosas.

Esto lo podemos programar nosotros

o

podemos usar la clase

`UIViewController`

que ya lo hace

La Clase UIViewController

- Es la **clase base** para crear nuestras propias clases VC, es decir, nuestras propias pantallas.
  - Crearemos **clases derivadas** para añadir la lógica de nuestra aplicación.
    - Sobreescribiendo o añadiendo propiedades y métodos.
- Proporciona la parte controladora (**C**) del patrón MVC.
- Proporciona una vista (**V**) vacía a la que añadiremos nuestra jerarquía de views.
  - La jerarquía de views se puede crear programáticamente, o
  - cargando el GUI desde un fichero storyboard o XIB.
    - Definiremos IBOutlets y IBActions para enganchar las propiedades y métodos entre el código y el diseño gráfico.
- **No** proporciona el modelo (**M**).
- Esta clase base nos proporciona también muchos métodos y propiedades para realizar las tareas descritas anteriormente: gestión de memoria, rotaciones, navegación, transiciones, etc...
  - Sobreescribiremos estos métodos para adaptarlos a nuestras necesidades.

La Clase UIViewController

- Al diseñar nuestra aplicación, diseñaremos todas las pantallas que la forman.
  - cada una de estas pantallas se encargará de una tarea
- En general, cada pantalla será un objeto de la clase **UIViewController**.
  - o mejor dicho, de una clase derivada de ésta.
- Para navegar entre las distintas pantallas usaremos controladores de navegación, controladores de pestañas, vistas modales, …
  - Los estudiaremos en otros temas.

Nota: una pantalla puede mostrar varias UIVC simultáneamente si se usa un SplitViewController o se usan vistas contenedoras.

Crear Ficheros VC.swift

- ¿Cómo se crea un fichero .swift que sea una subclase de UIViewController?
  - A mano escribiendo el fichero desde cero.
  - o usando una plantilla de Xcode:

    Xcode **>** *menú File* **>** *New* **>** *File* **>**
    iOS + Source **>** *Cocoa Touch Class* **>** *Next* **>**
    Poner nombre a la clase **>** *Subclass of UIViewController* **>**
    Con o sin XIB para el GUI **>** *¿Para iPhone, iPad o Universal?* **>**
    Lenguaje Swift **>** *Next* **>**
    Seleccionar Group (subdirectorio) donde se crearán los ficheros **>**
    Marcar Targets (dependencias de compilación) **>** *Create*

Crear un Proyecto

- Para crear un proyecto que sólo contiene una pantalla, es decir, una única clase UIViewController:

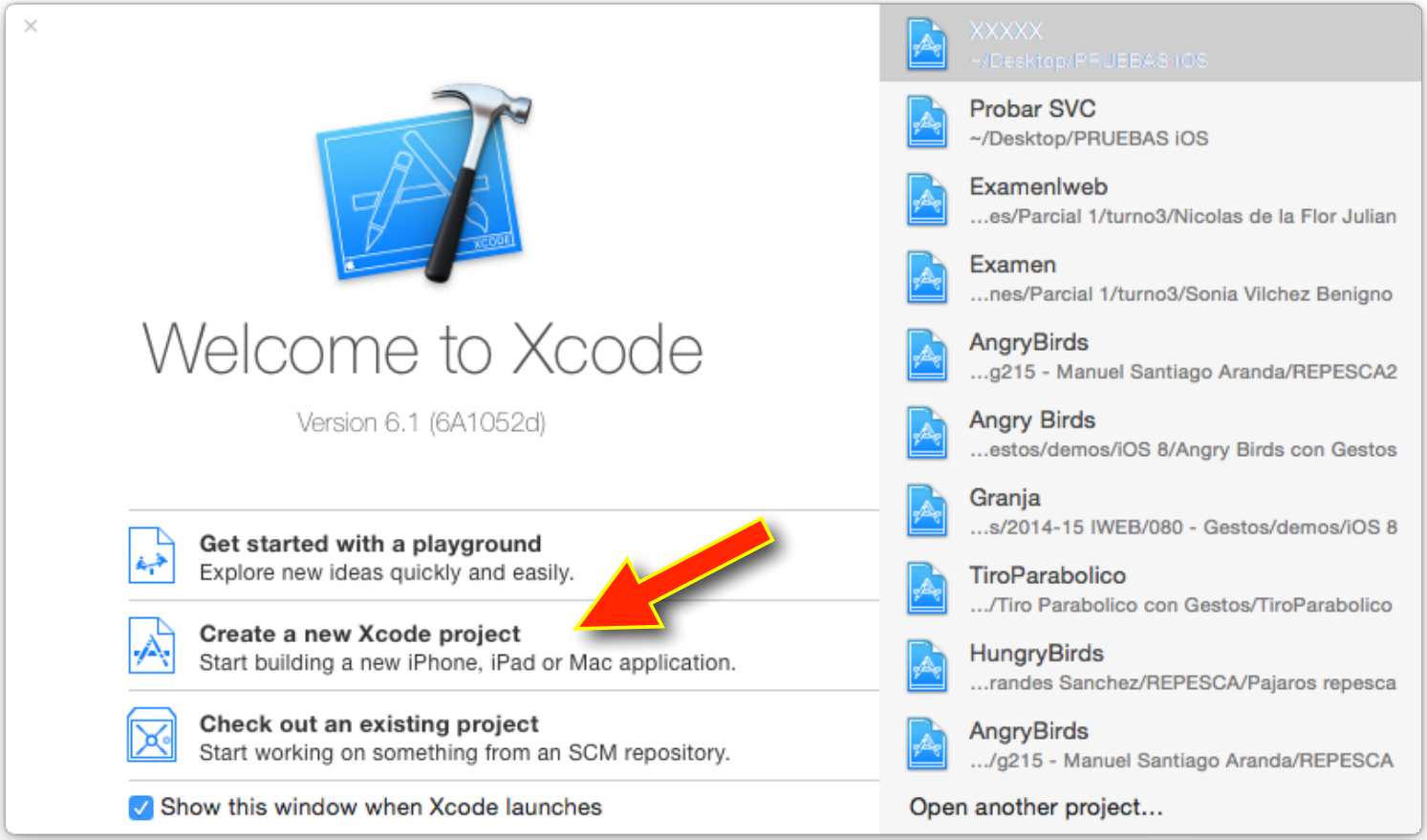
  Xcode **>** *menú File* **>** *New* **>** *Project* **>**
  iOS + Application **>** *Single View Application* **>** *Next* **>**
  Poner nombre al proyecto, a la organización, identificador de la organización (dominio del correo al revés), Lenguaje Swift, Tipo de Device (iPhone, iPad o Universal), ¿Usar core Data? **>** *Next* **>**
  Seleccionar subdirectorio donde se creará el proyecto **>** *Create*

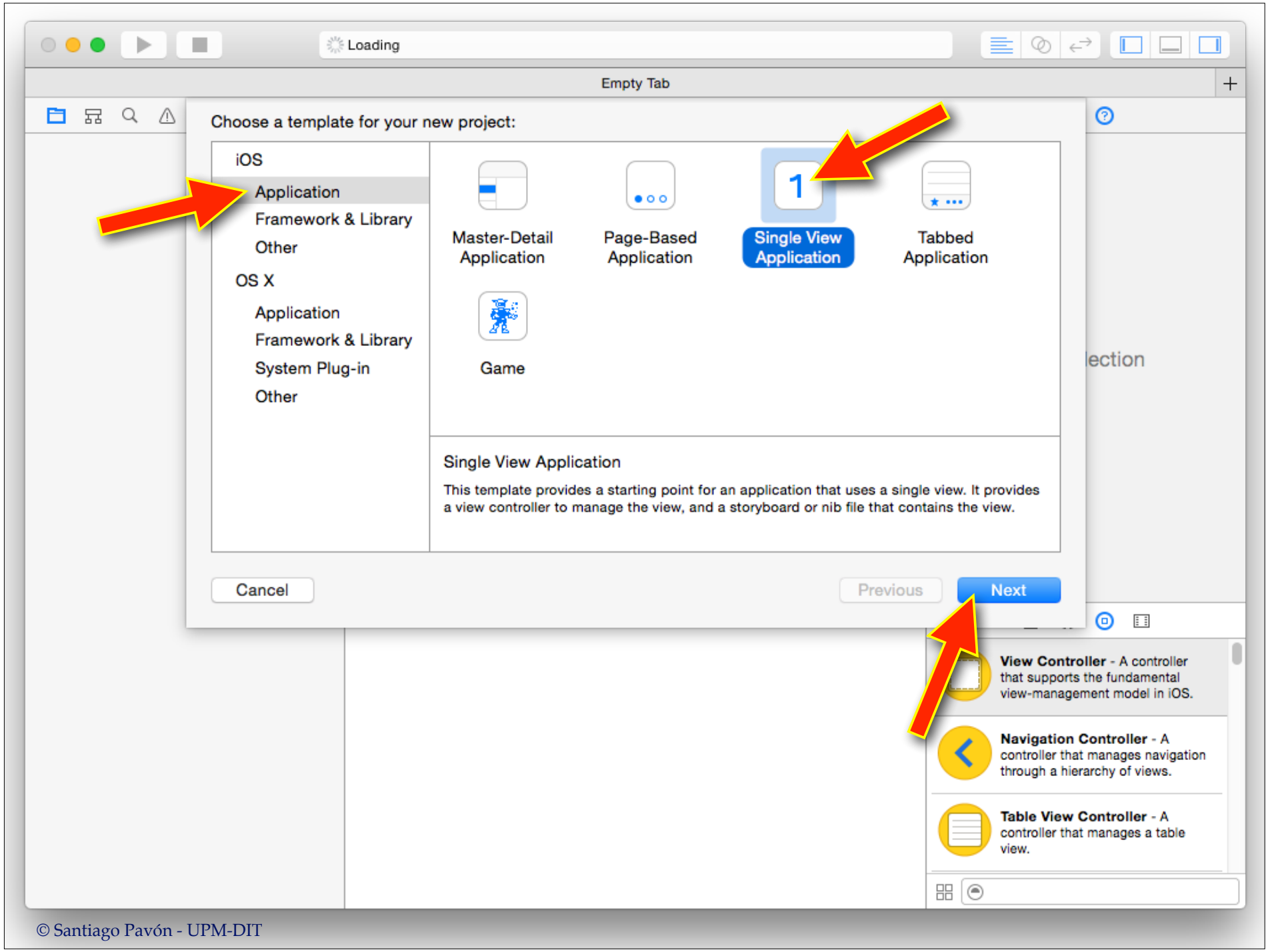
- Se crean los ficheros del proyecto (variará según opciones seleccionadas):

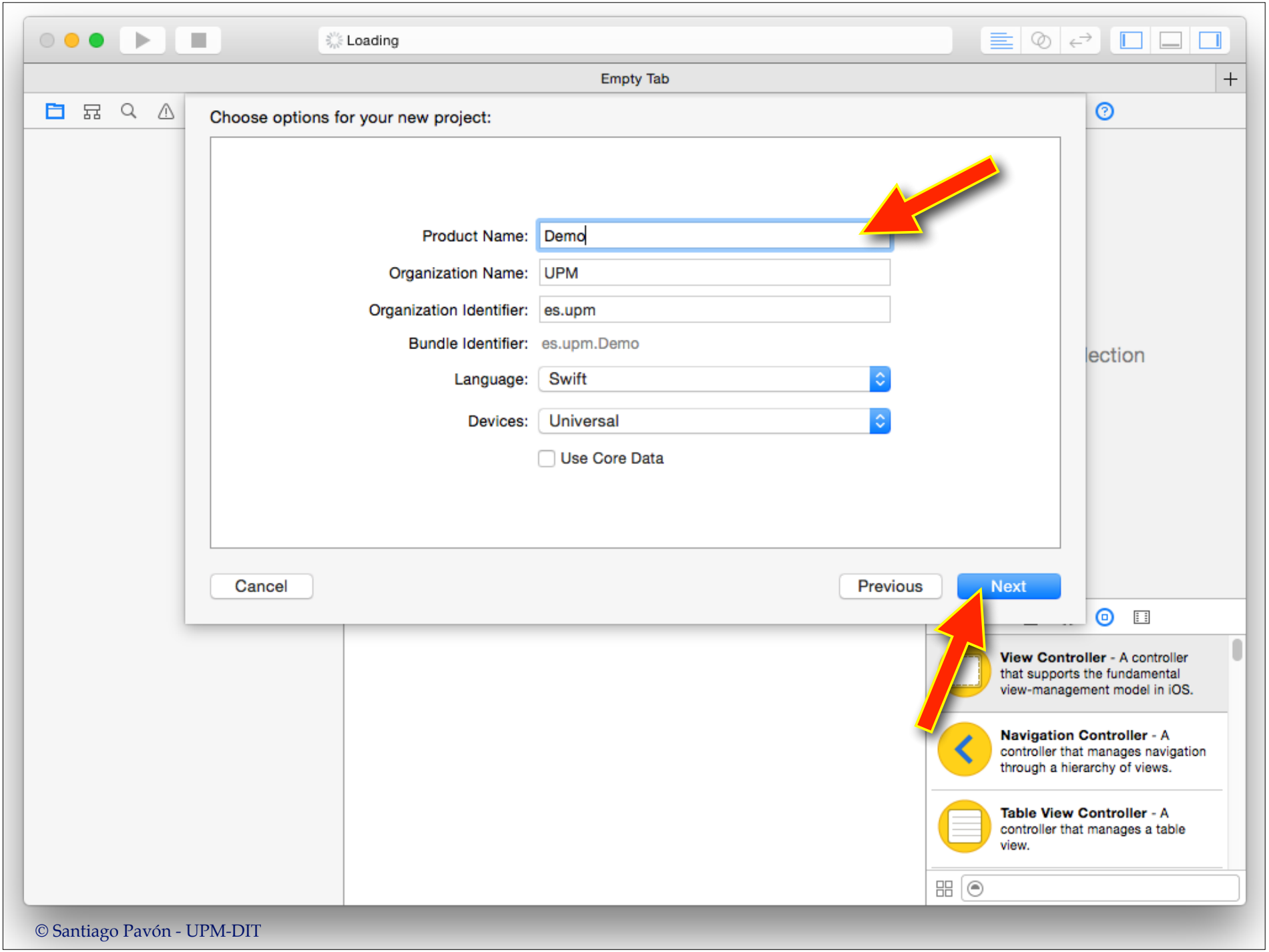
```
AppDelegate.swift
ViewController.swift
Main.storyboard
Images.xcassets
LaunchScreen.xib
Info.plist
. . .
```

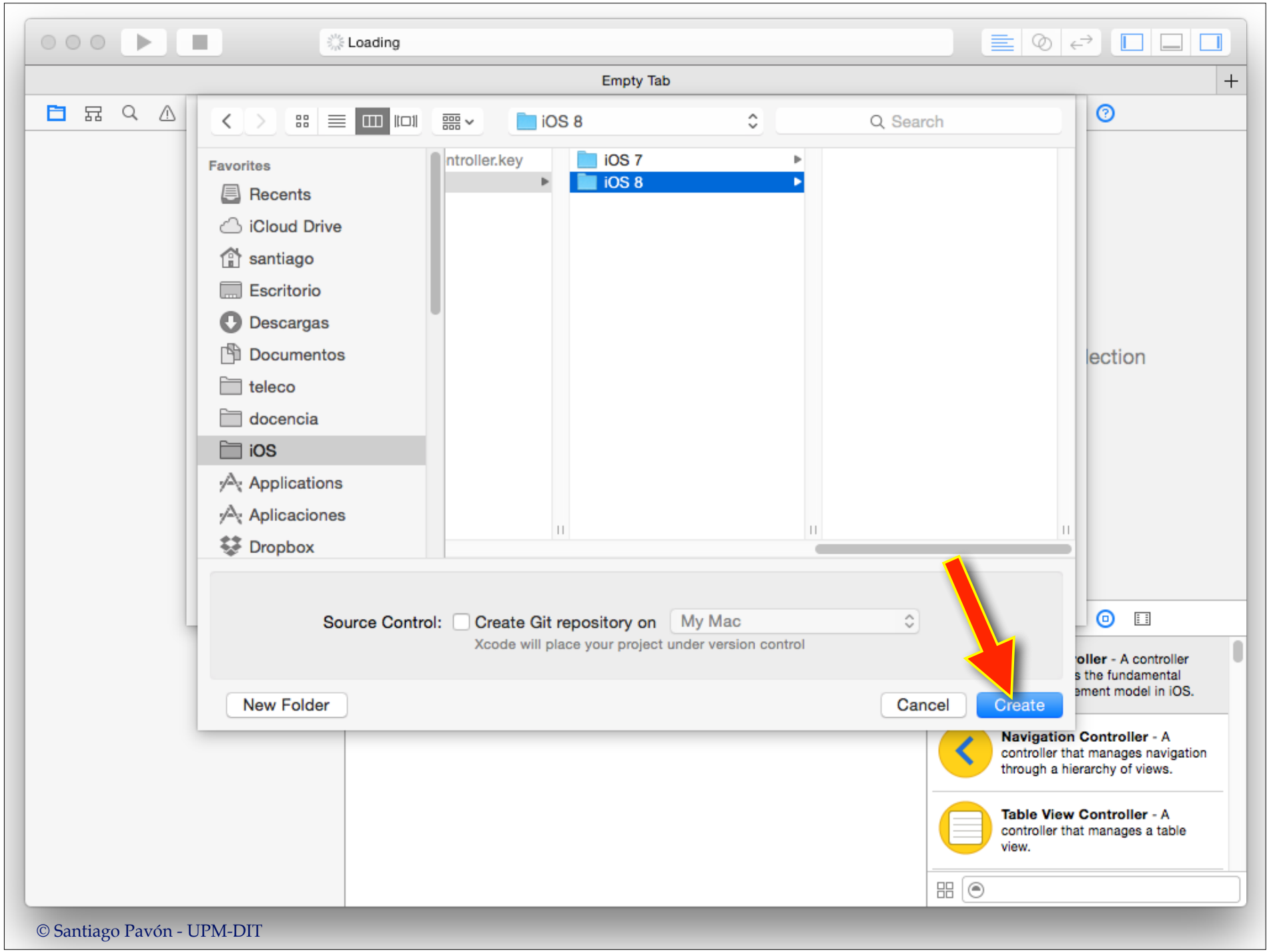
DEMO

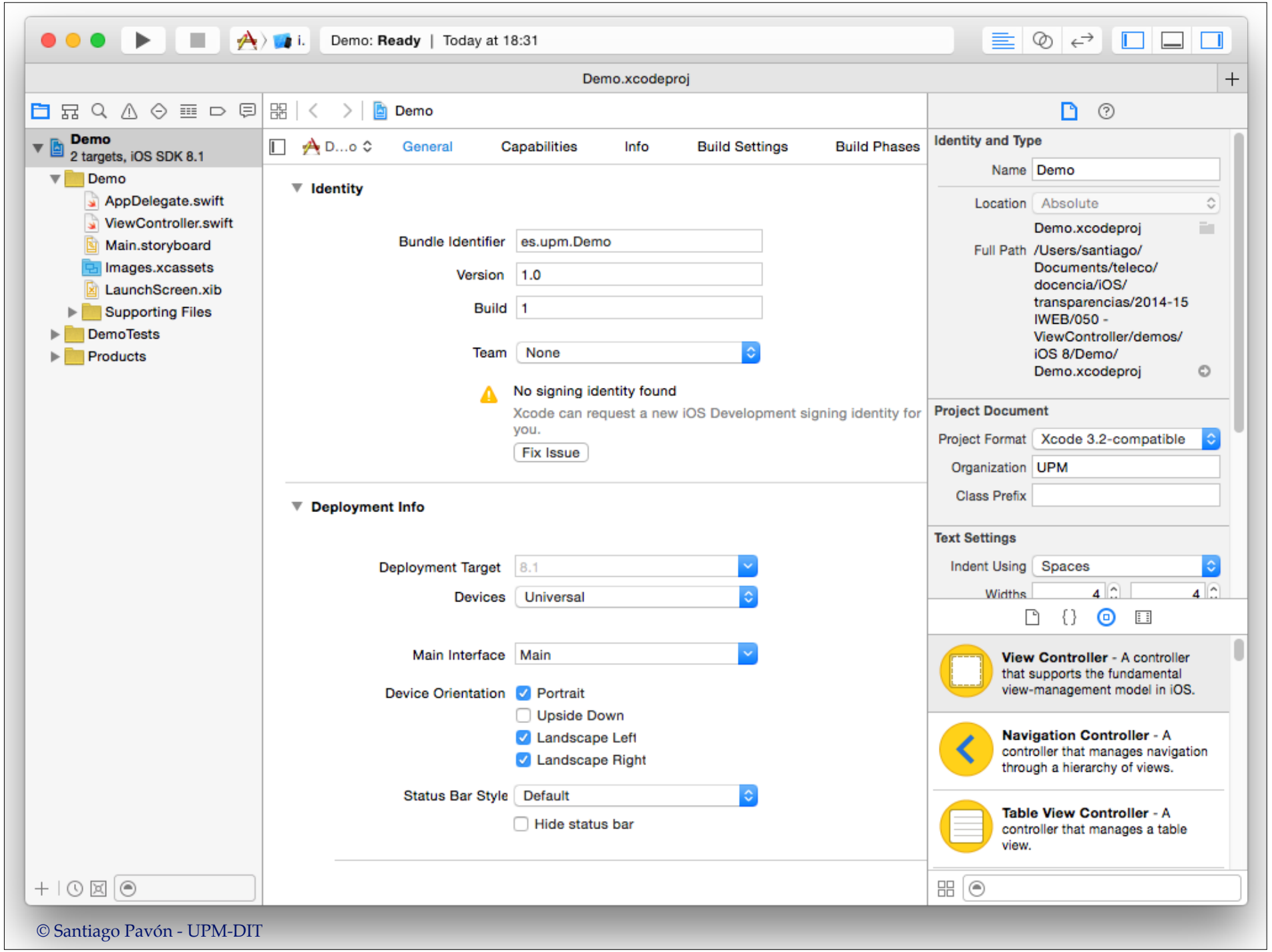
- Crear un proyecto llamado Demo usando la plantilla Single View Application

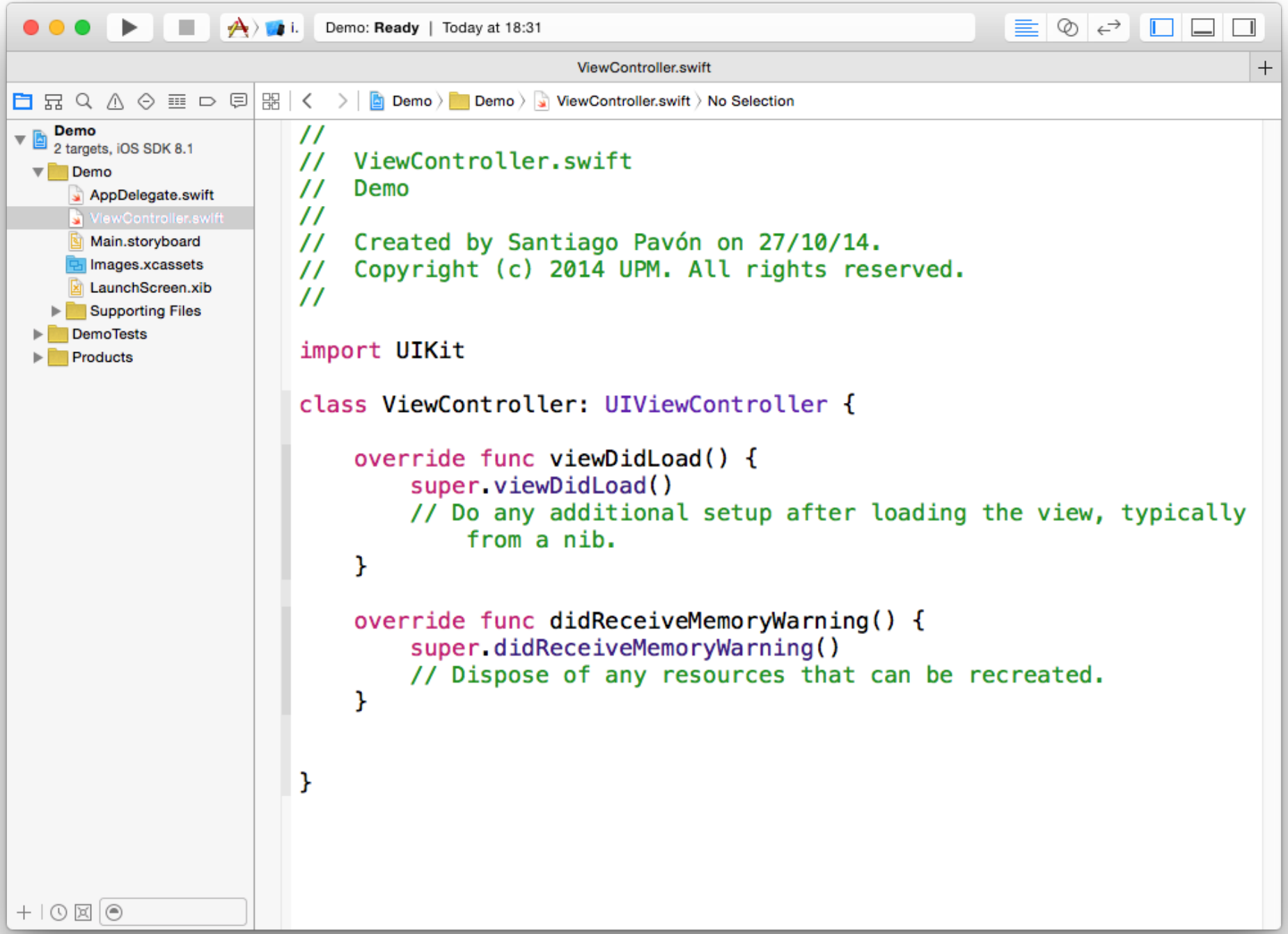


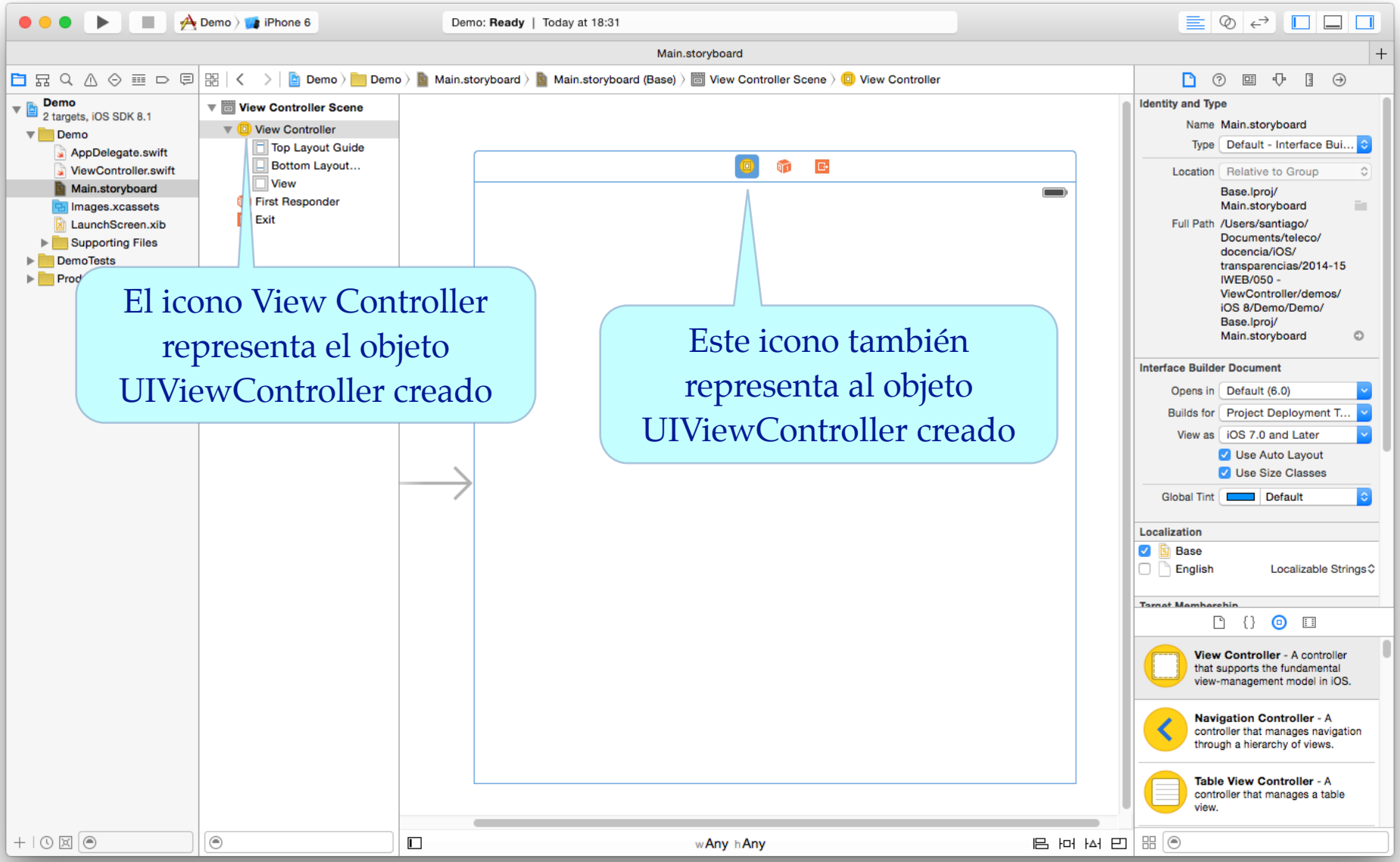


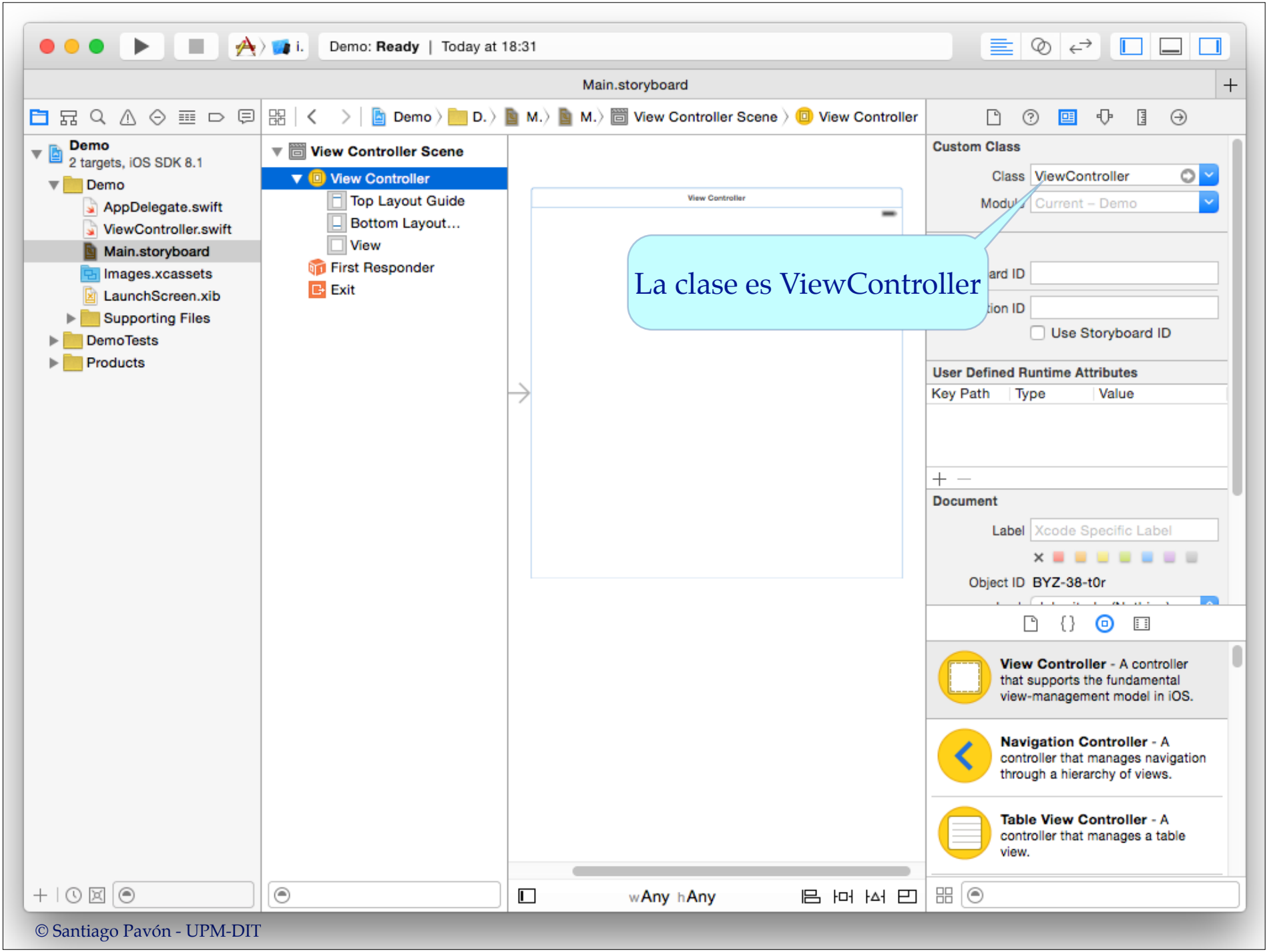


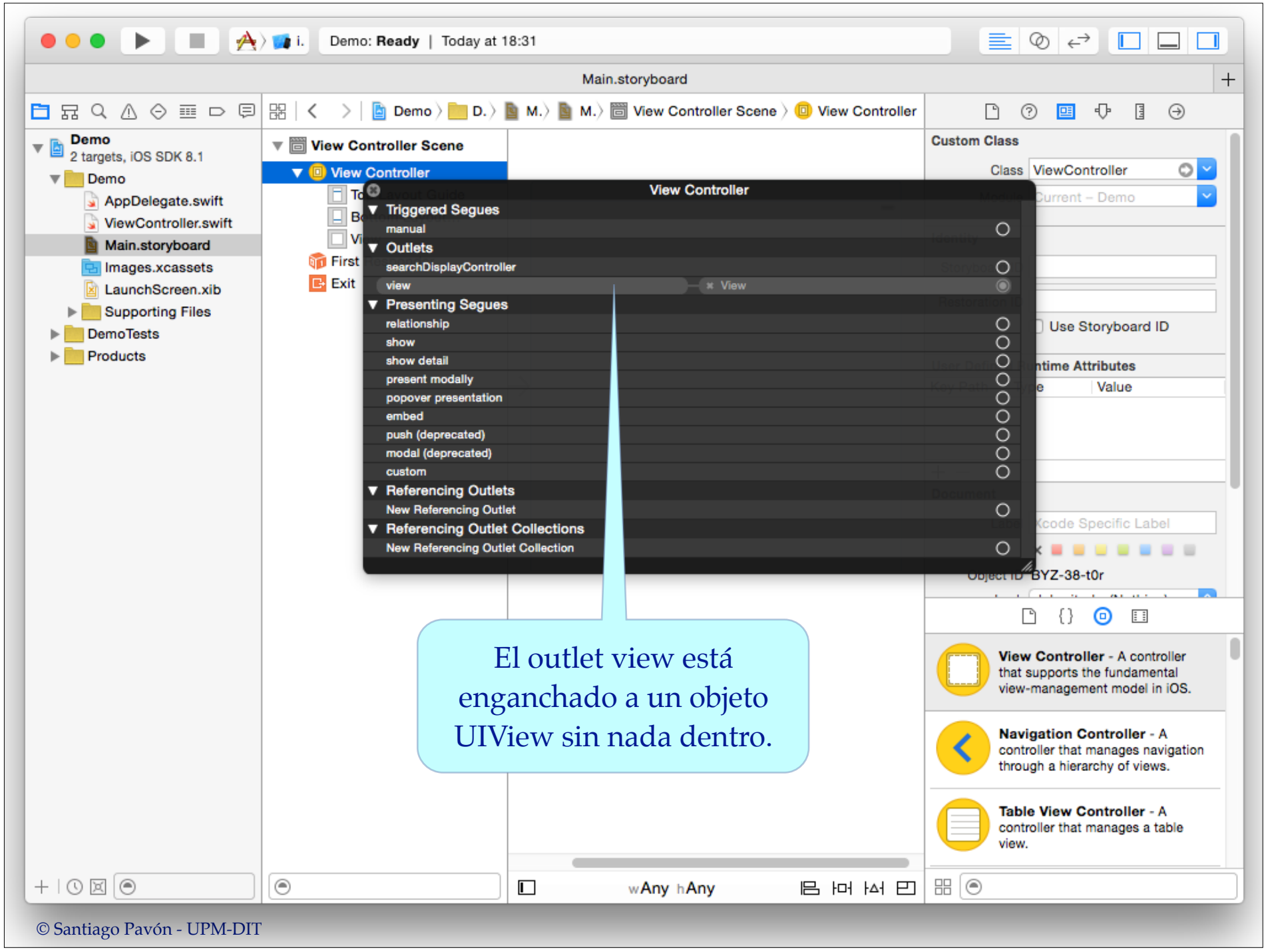












Inicialmente la pantalla no tiene nada.
Solo es una UIView vacía.

Propiedades y Métodos de UIViewController

view

- De la clase base UIViewController heredamos la propiedad **view**.

```
var view: UIView
```

  - Apunta a la raíz de nuestra jerarquía de views.

- Normalmente la jerarquía de views se carga desde un fichero storyboard (o XIB).

- Si no creamos la jerarquía de views desde un fichero storyboard (o XIB), entonces debemos crearla programáticamente.
  - El GUI se crea de forma perezosa:
    - Hasta que no consultamos por primera vez al valor de la propiedad **view** no se construye el GUI.
  - Si al acceder a la propiedad **view** su valor es **nil**, se llama automáticamente al método **loadView**.
    - Este método debe crear el GUI programáticamente y asignar un valor a la propiedad **view**.

- No retener subviews:
  - La propiedad **view** retiene a todos los objetos que forman parte de su jerarquía de subviews.
    - los apuntará directamente o a través de una subview intermedia
  - No es necesario que nuestros outlets también retengan a estas subviews.

```
@IBOutlet weak var label: UILabel!
```

  - Hay que evitar crear bucles de retenciones.

isViewLoaded

- Este método indica si la view está ahora mismo cargada en memoria.

```
func isViewLoaded() -> Bool
```

- A diferencia de la propiedad **view**, al llamar a este método no se intenta cargarla view en memoria, en caso de que no esté cargada actualmente en memoria.

viewDidLoad

- Este método se llama después de cargar la view (y sus subviews) en memoria, y de enganchar los outlets existentes.
  - tanto si la view se cargó desde un storyboard/XIB o se creó en loadView.

```
func viewDidLoad()
```

- Este método se usa típicamente para hacer inicializaciones que no pueden hacerse hasta que vista ya se ha cargado.
- Cuando se invoca este método, aun no se ha calculado la geometría de la vista.
  - En este método no pueden hacerse configuraciones que estén relacionadas con la geometría de las vistas.

- Ejemplo:
  - Crear un VC que muestre en una `UILabel` el texto guardado en una propiedad.

    - En el constructor (`init`???) del VC no se puede cambiar el texto de una `UILabel` del GUI.
      - Ya que la `UILabel` aun no existe aun, todavía no se ha cargado en memoria.

    - Sin embargo, el texto de la `UILabel` si puede cambiarse en el método `viewDidLoad`.
      - Cuando se llama a este método, la vista y sus subvistas ya han sido cargadas.
        - Sobreescribiremos el método `viewDidLoad` para asignar el valor que se desee como texto de la etiqueta.

```
override func viewDidLoad() {

    super.viewDidLoad()

    unaLabelDeLaVista.text = msg;
}
```

Muy importante:
No olvidar llamar al padre.

Ponemos como texto de la etiqueta el valor guardado en la propiedad **msg**.

El outlet **unaLabelDeLaVista** es UILabel que se carga desde un storyboard.
Al crear el objeto VC la view y sus subvista no se han cargado.
Cuando se llama a **viewDidLoad**, la vista ya ha sido cargada, y **unaLabelDeLaVista** ya existe. Ahora es cuando se puede poner como texto de la etiqueta el valor guardado en la propiedad **msg**.

awakeFromNib

- Este método se hereda de **NSObject**.
- Se usa para hacer configuraciones **después de haber cargado** un fichero XIB o una escena de un Storyboard.
- Los VC lo implementan para realizar configuraciones que no pueden hacerse hasta que se han creado **todos** los objetos definidos en un fichero Interface Builder.
  - Se invoca en todos los objetos creados al cargar un fichero XIB o una escena de un Storyboard.
  - Se invoca después de que todos los objetos del fichero XIB o escena del Storyboard ya han sido creados, y todos los outlets y conexiones se han realizado.
- No olvidar llamar a la versión de este método tapada de la superclase.

Cambios en la Visibilidad del VC

- Avisos indicando que el objeto ViewController se va a hacer visible, que ya es visible, que va a ocultarse, o que ya se ha ocultado.

```
func viewWillAppear(_ animated: Bool)
func viewDidAppear(_ animated: Bool)
func viewWillDisappear(_ animated: Bool)
func viewDidDisappear(_ animated: Bool)
```

- Sobreescribir estos métodos si queremos hacer algo en estos instantes.
  - Por ejemplo, gestionar la persistencia de algún valor, refrescar el dato mostrado por alguna view, …
  - No olvidar llamar a la versión tapada de la clase padre (**super.???**).
- Estos métodos se llaman cada vez que cambia la visibilidad del VC.
  - Ocurrirá con frecuencia en las aplicaciones que tienen varias pantallas.
    - Cuando la aplicación termina no se llama a XXXDisappear.
- Cuando se invoca a **viewWillAppear** la geometría de las vistas ya se ha calculado.

Geometría de la Vistas

- Cuando se invoca a **viewDidLoad** aun no se ha calculado la geometría de las vistas.
  - El valor de las propiedades **bounds** y **frame** no se ha calculado.
  - Dentro de viewDidLoad no podemos hacer cambios que dependan de las geometrías.
- Cuando se invoca a **viewWillAppear** la geometría de las vistas ya se ha calculado.
- Cuando cambia el valor de la propiedad **bounds** o **frame** de una vista, ésta reajusta la posición de todas sus subviews.
  - Pero antes de reposicionar las subviews se llama al método:

```
func viewWillLayoutSubviews()
```

  - Y después de reposicionar las subviews se llama al método:

```
func viewDidLayoutSubviews()
```

  - Redefinir estos métodos cuando queramos hacer cambios relacionados con la geometría de las vistas.
    - Por ejemplo, al girar la pantalla.

- Otro método que se puede sobreescribir para añadir o eliminar restricciones de autolayout es:

```
func updateViewConstraints()
```

  - No olvidar llamar a la versión tapada de la clase padre.

Orientación del Terminal

- Las orientaciones soportadas se indican en **Info.plist**, pero normalmente se seleccionan gráficamente en Xcode en las propiedades del target a construir.
- Pero un VC puede modificar estos valores sobreescribiendo el método

```
func supportedInterfaceOrientations() -> Int
```

  - Es una mascara de bits formada con los valores del enumerado **UIInterfaceOrientationMask**.

- Para indicar cual es la orientación preferida para presentar inicialmente el VC hay que sobreescribir el método:

```
func preferredInterfaceOrientationForPresentation()
        -> UIInterfaceOrientation
```

- Para indicar si el contenido de un VC debe rotar hay que sobreescribir el método:

```
func shouldAutorotate() -> Bool
```

- En iOS 8 todos los métodos relacionados con la rotación del terminal se han quedado obsoletos (deprecados):

```
func willRotateToInterfaceOrientation(_:,duration:)

func willAnimateRotationToInterfaceOrientation(_:, duration:)

func didRotateFromInterfaceOrientation(_:)
```

- En iOS 8 las rotaciones se tratan como un cambio del tamaño de la view de los ViewControllers.
  - Cuando rota un terminal y cambia el tamaño de la view se llama al método:

```
func viewWillTransitionToSize(_ size: CGSize,
    withTransitionCoordinator coordinator:
                              UIViewControllerTransitionCoordinator)
```

  - Sobreescribirlo para realizar los cambios que se deseen.
  - No olvidar llamar a la versión tapada de la clase padre.

- En iOS 8, si al rotar un terminal hay cambios en el TraitCollection de un ViewController (también se llama en las UIViews afectadas), se llama a los métodos:

```
func willTransitionToTraitCollection(_ newCollection: UITraitCollection,
          withTransitionCoordinator coordinator:
                                    UIViewControllerTransitionCoordinator)

func traitCollectionDidChange(_ previousTraitCollection: UITraitCollection?)
```

  - Sobreescribirlos para realizar los cambios que se deseen.
  - No olvidar llamar a la versión tapada de la clase padre.

Notas

ANTIGUO:

- viewDidUnload ha sido deprecado en iOS6.
  - También ha sido deprecado viewWillUnload.

- Este método se invocaba cuando había problemas de falta de memoria en la aplicación.
  - Se invocaba en los VC que no estaban visibles en ese momento para que liberarán toda la memoria que pudieran.
  - Si el VC se hacía visible otra vez en el futuro, se volvía a llamar a viewDidLoad que regeneraría los datos liberados.
  - En viewDidUnload debía liberarse todo aquello que pudiera reconstruirse más tarde en viewDidLoad.

- Ahora las liberaciones de memoria se hacen en:
  - didReceiveMemoryWarning.

didReceiveMemoryWarning

- Este método se llama cuando hay problemas de falta de memoria.

```
func didReceiveMemoryWarning()
```

- Solo debe liberarse memoria de un VC cuando **no se esté mostrando** en la pantalla.

```
override func didReceiveMemoryWarning() {

    super.didReceiveMemoryWarning()

    if isViewLoaded() && view.window == nil {

        // Liberar memoria que pueda regenerarse otra vez
        // en un futuro en viewDidLoad o viewWillAppear, …
        mi_tabla_de_fotos = nil
    }
}
```

- Aclaraciones sobre el ejemplo anterior:

- Primero debe llamarse a la versión del método tapada en el padre.

```
super.didReceiveMemoryWarning()
```

- La sentencia **if** comprueba que el VC no sea visible en la pantalla en este momento.
  - Primero hay que comprobar que **view** esté cargada en memoria:

```
isViewLoaded()
```

    - Esta comprobación es necesaria porque si **view** no está cargada en memoria, se cargaría automáticamente de nuevo al acceder a **view** al evaluar la expresión **view.window**.

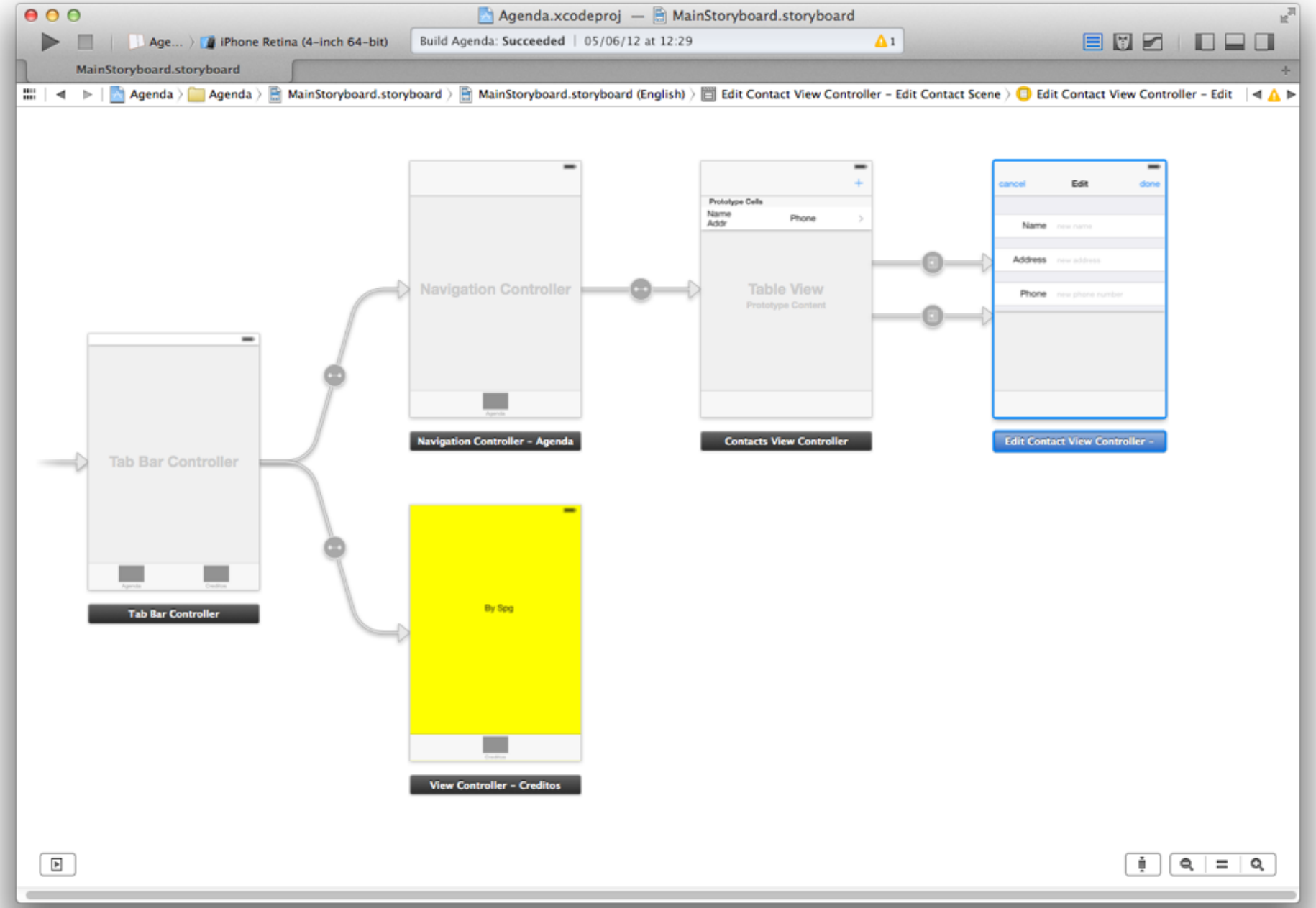
  - Un VC no está visible en pantalla si la propiedad **window** de su propiedad **view** es nil.

```
view.window == nil
```

- Si la condición del **if** se cumple, se liberan todos aquellos objetos que puedan regenerarse en un futuro en caso de necesidad.
  - Si este VC vuelve a hacerse visible, se llamará otra vez a **viewDidLoad**, donde se regenerarían los objetos liberados aquí.

Crear VC usando Storyboard

- Lo más normal es que las aplicaciones usen un storyboard para diseñar sus pantallas o escenas,
  - y las conecten y relacionen entre sí usando segues.
- Estas pantallas o escenas serán:
  - objetos View Controller y
  - controladores de navegación.
- Los objetos VC diseñados en el storyboard se instanciarán **automáticamente** cuando se necesiten.
  - Los segues existentes indican que VC deben instanciarse.
  - y sólo necesitamos configurarlos en **prepareForSegue:sender:**.
- No es muy normal crear los objetos VC programáticamente.



- Para crear **programáticamente** un VC definido en un fichero storyboard:

  - Primero hay que obtener el objeto storyboard.
    - Puede obtenerse:
      - desde un VC ya existente accediendo a su propiedad **storyboard**.
      - creándolo desde un fichero **.storyboard** con el siguiente inicializador de la clase **UIStoryboard**:

```
init(name name: String,
     bundle storyboardBundleOrNil: NSBundle?) -> UIStoryboard
```

  - Segundo, al objeto storyboard que hemos obtenido en el punto anterior, le pedimos que instancie un objeto VC usando el método:

```
func instantiateViewControllerWithIdentifier(
                    _ identifier: String) -> AnyObject!
```

    - El parámetro indica cual de los VC definidos en el storyboard es el que queremos instanciar.
      - Estos identificadores se crean con el inspector de atributos del Interface Builder.

    - También podemos instanciar el VC inicial del storyboard con este método:

```
func instantiateInitialViewController() -> AnyObject
```

Instanciar programáticamente un VC desde un storyboard:

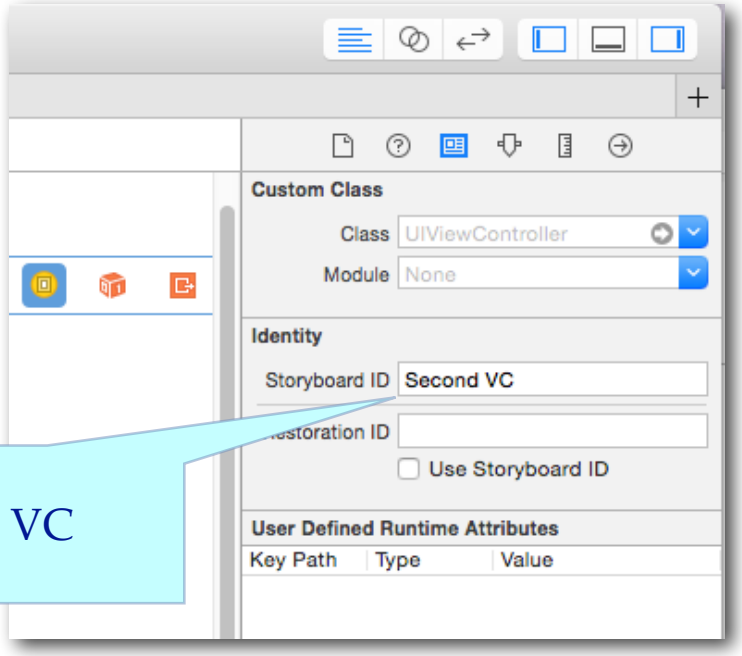
El VC actual se creó desde el storyboard apuntado por esta propiedad.

```
if let vc =
    storyboard?.instantiateViewControllerWithIdentifier("Second VC")
                        as? UIViewController {

    presentViewController(vc, animated: true, completion: nil)

}
```

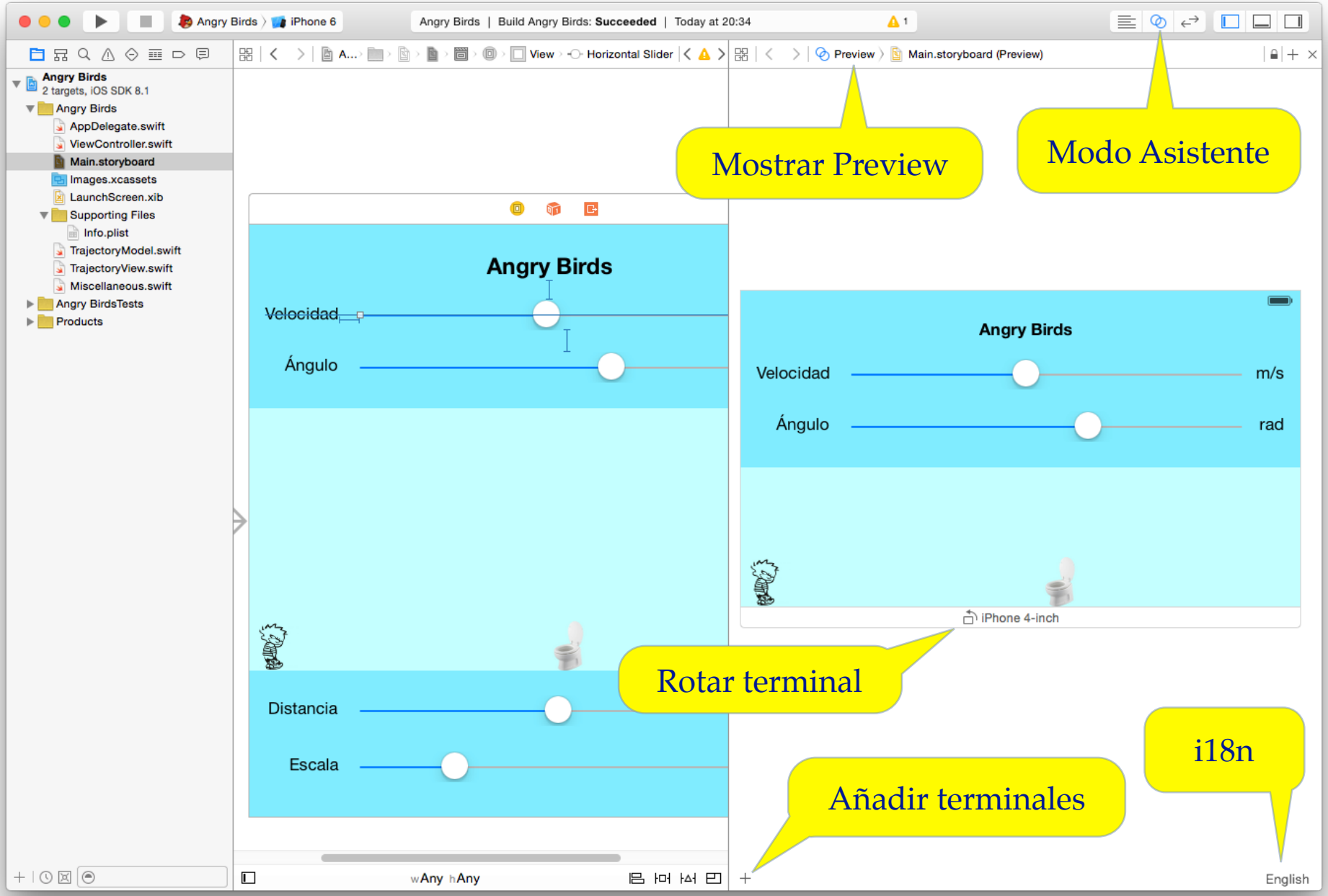
Desde el VC actual muestro el VC recién creado de forma modal



Identificador del VC

Previsualización de Pantallas

- Con el Interface Builder podemos previsualizar cómo quedan las pantallas (*los View Controles*) diseñados para distintos modelos de terminal, distintas orientaciones del terminal, diferentes idiomas, …



Crear VC usando XIB

- Para crear programáticamente una instancia VC que cargue un fichero XIB:
  - usamos el siguiente inicializador de UIViewController.

```
init(nibName nibName: String?,
     bundle nibBundle: NSBundle?)
```

- Usa el XIB y el bundle especificado.

Instanciar programáticamente un VC usando un XIB:

```
class ModalViewController: UIViewController {

    override init(nibName nibNameOrNil: String?,
                  bundle nibBundleOrNil: NSBundle?) {

        super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)

        // inicializaciones varias
    }
   . . .
}
```

```
let modalVC = ModalViewController(nibName: "ModalViewController",
                                  bundle: nil)

presentViewController(modalVC, animated: true, completion: nil)
```

Relaciones entre VC

Aplicaciones complejas

- Muchas aplicaciones están formadas por varias pantallas.
  - Cada pantalla será un VC que proporcionará una determinada funcionalidad al usuario.
  - Se mostrará una pantalla u otra según las acciones realizadas.
- ¿Cómo se crean aplicaciones con varias pantallas?

Controladores de ViewControllers

- Existen VC controladores que manejan otros VC:
  - Crean su vista usando las vistas de otros VC.

➡**`UINavigationController`**

- Maneja una pila de VC.

➡**`UITabBarController`**

- Selección de VC independientes usando pestañas.

➡**`UISplitViewController`**

- VC maestro que controla los detalles mostrados en otro VC.

- etc . . .

ViewControllers Modales

- Un ViewController puede mostrar de forma modal otro VC.
  - En iPhone los VC modales ocupan toda la pantalla
  - En iPad pueden mostrarse con diferentes estilos.

Relaciones entre Controladores

- Los objetos ViewController poseen propiedades para acceder a los VC con los que están relacionados

```
tabBarController
navigationController
parentViewController
presentingViewController
presentedViewController
splitViewController
popoverPresentationController
presentationController
childViewControllers
```