POLITÉCNICA

ETSIT UPM

dit UPM

Desarrollo de Apps para iOS

Anatomía de una Aplicación

IWEB 2015-2016
Santiago Pavón

ver: 2015.10.05

© Santiago Pavón - UPM-DIT

Anatomía de una Aplicación

- Una aplicación está compuesta de:
  - El código generado por el programador.
  - Frameworks usados:
    - Foundation, UIKit, MapKit, . . .
      - Proporcionan las clases base a usar, recursos, ...
  - Ficheros Storyboard, NIB.
  - Recursos:
    - catálogo de activos (xcassets)
    - imágenes, iconos, sonidos, . . .
  - Ficheros de configuración:
    - Info.plist
  - . . .

© Santiago Pavón - UPM-DIT

Ciclo de Vida de una Aplicación

- UIKit crea la aplicación y controla su ciclo de vida.
  - Crea una instancia UIApplication.
    - e instancias de otras clases.
  - Carga fichero storyboard principal.
  - Atiende los eventos.
  - Gestiona la barra de estado.
  - Manejo de las interrupciones.
  - etc.
- Durante la ejecución de una aplicación se invocan muchos métodos:
  - Unos pertenecientes a las clases creadas por el usuario.
  - Y otros pertenecientes a las clases proporcionadas por el SDK.
    - El programador debe sobrescribir algunos de estos métodos para programar el comportamiento deseado de la aplicación.
      - Deben crearse subclases de las clases proporcionadas por el SDK
      - y sobrescribir los métodos que interese.

© Santiago Pavón - UPM-DIT

El Código de la Aplicación

- Una aplicación se crea partiendo del código generado por las plantillas proporcionadas por Xcode.
  - Generan un primer esqueleto con los ficheros necesarios de la aplicación.
    - Y luego añadiremos nuestro código para implementar las funcionalidades propias de nuestra aplicación.
    - Editaremos con el Interface Builder el fichero storyboard para crear parte del GUI.
    - Editaremos y configuraremos los ficheros de soporte generados por Xcode.
    - Etc.

© Santiago Pavón - UPM-DIT

Ficheros de Soporte

- Fichero de propiedades de la aplicación **`Info.plist`**:
    - orientaciones soportadas de la aplicación.
    - aplicación para iPhone.
    - nombre del fichero storyboard.
    - localización.
    - identificador de bundle.
    - . . .
  - Normalmente no se editan directamente los valores del plist, sino seleccionando el proyecto o el target, y usando los paneles mostrados por el editor de Xcode.
- Internacionalización: **`InfoPlist.strings`**.
- . . .

© Santiago Pavón - UPM-DIT

UIApplication

- Todas las aplicaciones tienen un único objeto **UIApplication** desde donde se controlan.
- Es un singleton al que se accede con:

```
UIApplication.sharedApplication()
```

  - No se crean clases derivadas de UIApplication.
    - podría hacerse, pero sólo para hacer cosas poco habituales.

- UIApplication usa un delegado para ajustar el comportamiento de la aplicación ante ciertos eventos.
  - Este delegado es una subclase de **`UIApplicationDelegate`**.
    - Esta clase se etiqueta con **`@UIApplicationMain`**.
  - Las plantillas de Xcode generan una subclase de UIApplicationDelegate que podemos modificar para adaptarla a nuestras necesidades.

© Santiago Pavón - UPM-DIT

UIApplicationDelegate

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

  var window: UIWindow?

  func application(application: UIApplication!,
       didFinishLaunchingWithOptions launchOptions: NSDictionary!)
                  -> Bool {}

  func applicationWillResignActive(application: UIApplication!) {}

  func applicationDidEnterBackground(application: UIApplication!) {}

  func applicationWillEnterForeground(application: UIApplication!) {}

  func applicationDidBecomeActive(application: UIApplication!) {}

  func applicationWillTerminate(application: UIApplication!) {}

  . . .
}
```

© Santiago Pavón - UPM-DIT

UIApplication Delegate Messaging
with Multitasking

V1.3

Start Here

App Starts

application:didFinishLaunchingWithOptions:

replaced previous didFinishLaunching

URL

Yes

No

application:openURL: sourceApplication:annotation:

applicationDidBecomeActive:

No

Yes

URL

Home Button or Task Switcher

Foreground Run Loop

Interruption, e.g. Phone Call

applicationWillResignActive:

applicationWillResignActive:

applicationDidEnterBackground:

Yes

User picks up

No

no longer a termination reason

applicationWillEnterForeground:

App Relaunched (e.g. Springboard, App Switcher or URL)

Background Run Loop

SDK >= 4

No

Yes

Yes

App Suspended in RAM

No

info.plist setting or task completion

No

UIApplication-ExitsOnSuspend

"opt out"

Yes

applicationWillTerminate:

OS needs to reclaim memory

Background

kill

App Ended

App has left the RAM

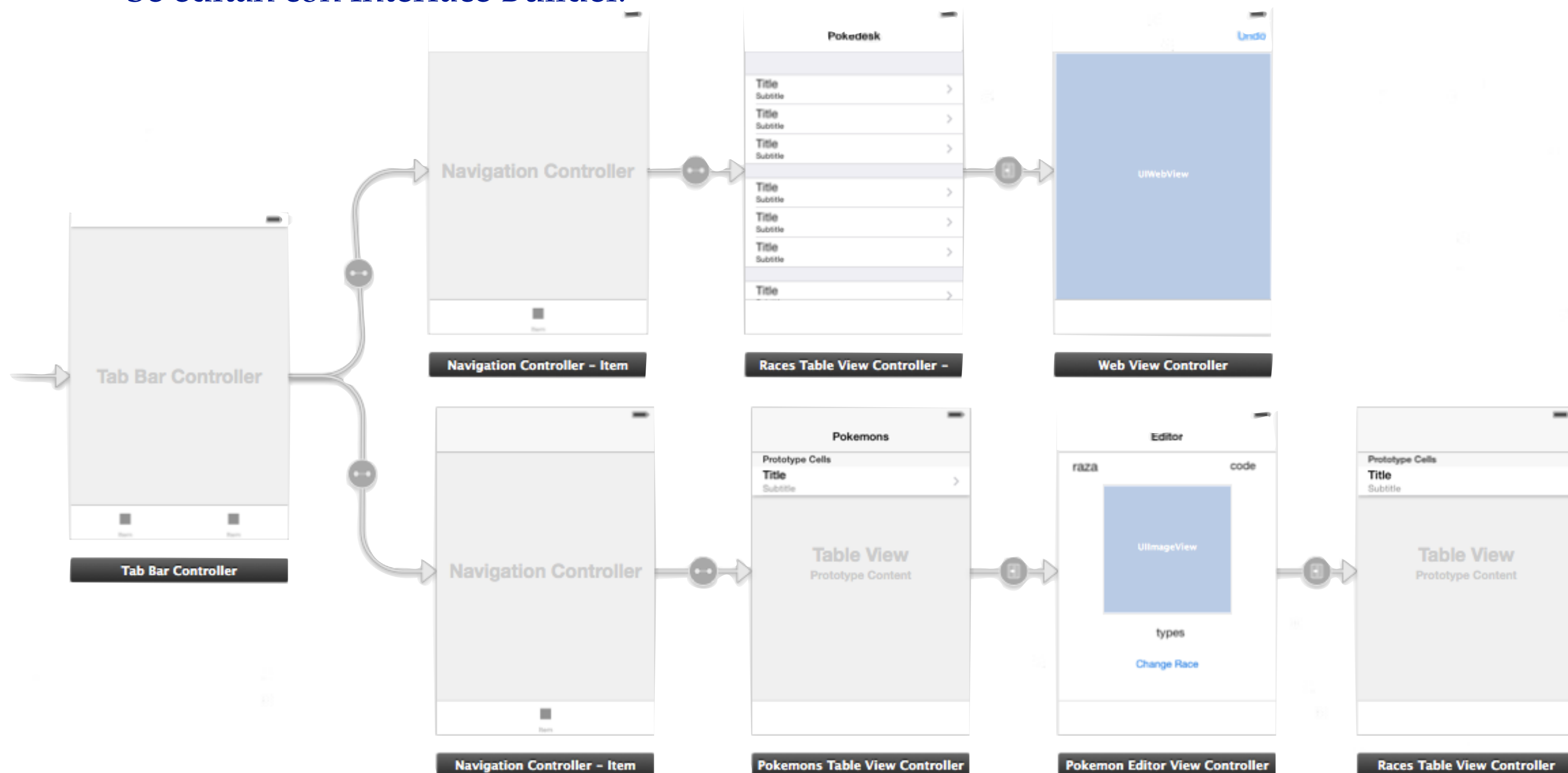
iOS4

© Santiago Pavón - UPM-DIT

created by Oliver Drobnik, oliver@cocoanetics.com, Twitter @cocoanetics

Ficheros Storyboard

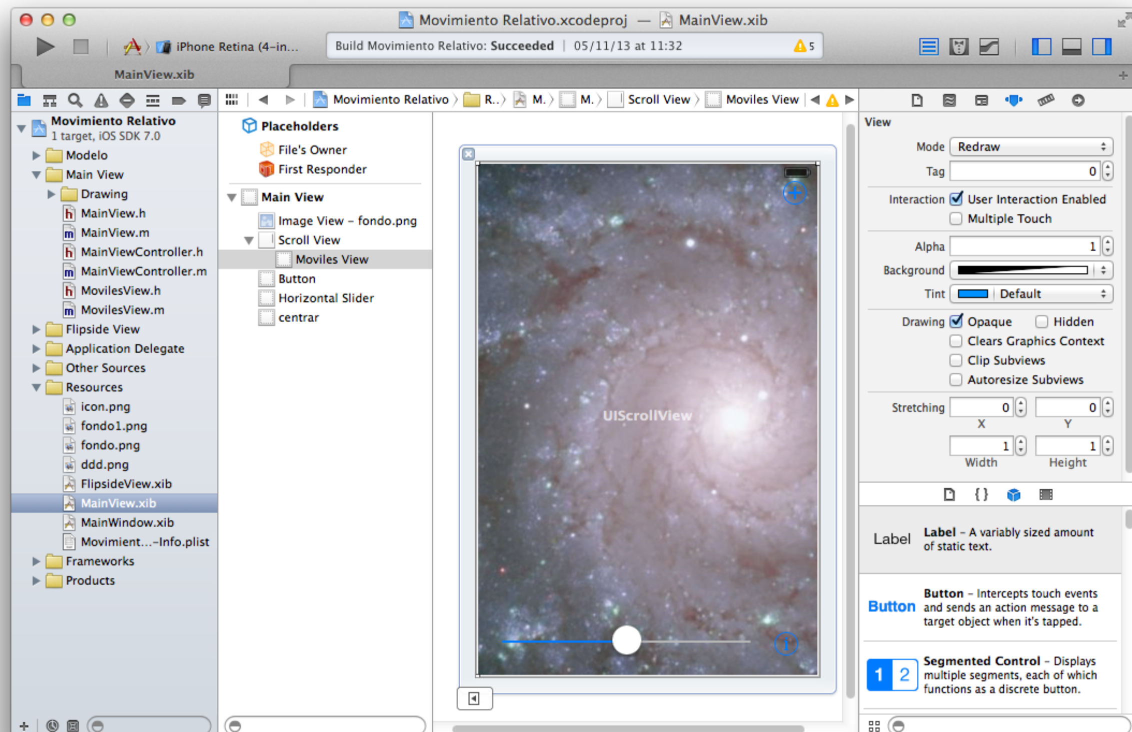
- El diseño de las pantallas (*su contenido, IBOutlets, IBActions, Adaptabilidad*) y la conexión entre ellas se hace principalmente usando ficheros storyboard.
    - Se editan con Interface Builder.



© Santiago Pavón - UPM-DIT

Ficheros XIB

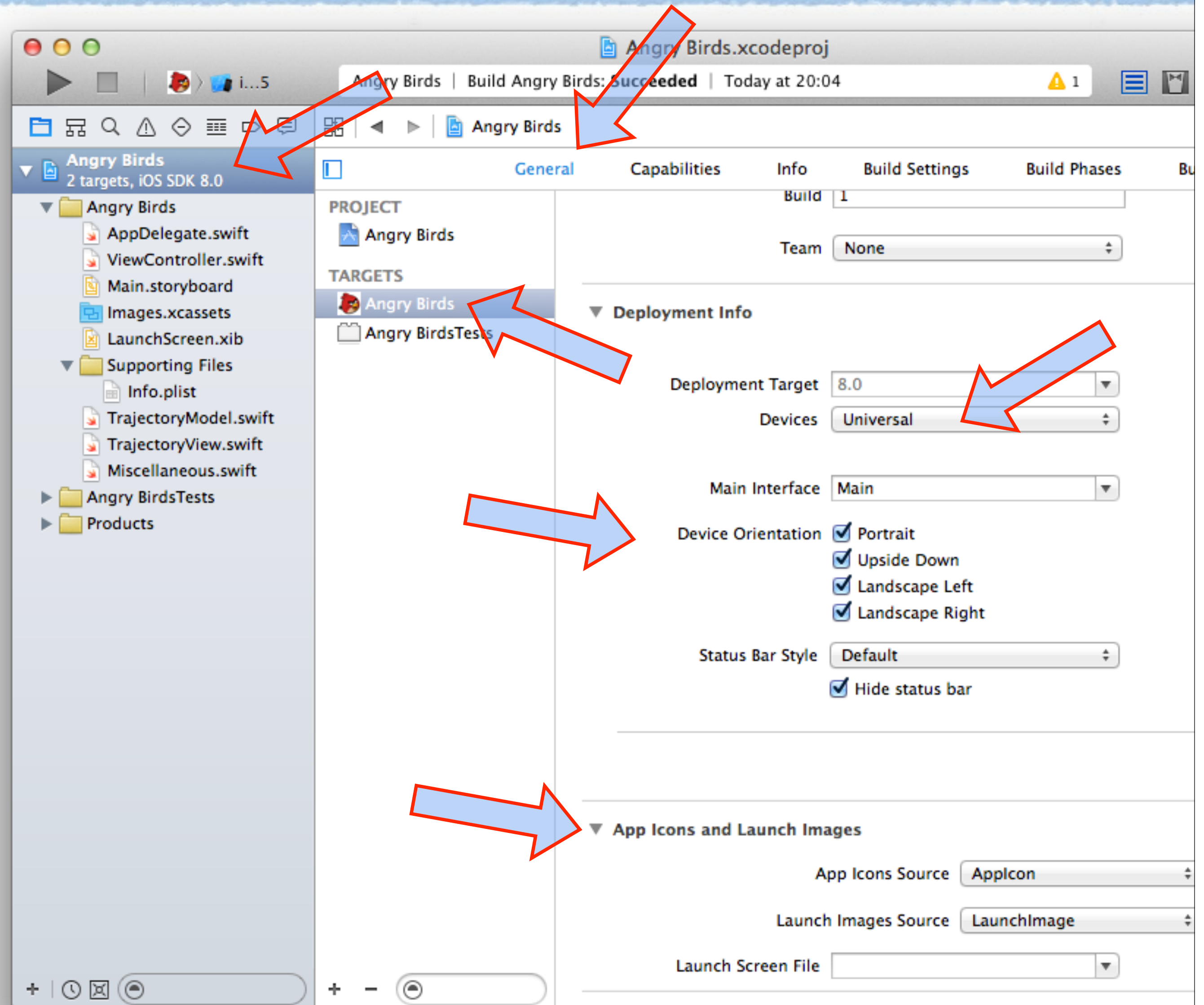
- Crear las pantallas del interface y otras views (celdas personalizadas de tablas, ...).
- Cada icono es una única instancia (un único objeto).
  - Orden de creación sin definir.
- Editados gráficamente con Interface Builder.
  - Crear objetos arrastrándolos desde la librería de objetos.
  - Conectar outlets y actions.
  - Inspector: Editar propiedades.
  - ...
- xib = nib + xml



© Santiago Pavón - UPM-DIT

Iconos, Orientación, Storyboard, ...

- Desde Xcode pueden configurarse muchos detalles de la app:
  - Iconos.
  - Imágenes de lanzamiento.
  - Nombre de Storyboard a usar para cada tipo de dispositivo.
  - Tipo de dispositivo.
  - Orientaciones permitidas del dispositivo.
  - . . .



© Santiago Pavón - UPM-DIT

La Interface de Usuario

- Formada por una **jerarquía de UIViews**:
  - La **raíz** es de la clase **UIWindow**.
    - Las plantillas de Xcode la crean por defecto.
  - **Debajo** de UIWindow se añaden **etiquetas**, **botones**, **tablas**, **views**, . . .
    - Son subclases de **UIView**, **UIControl**.
- La podemos crear la jerarquía de views usando Interface Builder o escribiendo el código directamente.

© Santiago Pavón - UPM-DIT

Las **`UIViews`**

- Áreas rectangulares.
- Manejan eventos.
- Dibujan su contenido.
- Propiedades:

  `alpha, hidden, frame, bounds, center, tag, contentMode, transform, superview, subviews, userInteractionEnabled, window, …`

- También tienen sus propios métodos, delegados, etc.
- Existen muchas predefinidas:

  `UIView, UILabel, UIButton, UISlider, UIImageView, …`

© Santiago Pavón - UPM-DIT

Creación Manual de un GUI

- Un ejemplo: añadir en un `UIViewController`:

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let r = CGRectMake(10, 10, 100, 100)

        let l = UILabel(frame: r)

        l.text = "hola"
        l.transform = CGAffineTransformMakeRotation(0.5)

        view.addSubview(l)
    }
}
```

© Santiago Pavón - UPM-DIT

Controles, Eventos, Acciones

- **Control**: Es una view que al ser manipulada por el usuario, genera eventos que provocan que se invoquen acciones en otros objetos.
  - Jerarquía: UIResponder -> UIView -> UIControl -> ???
  - Existen muchos controles predefinidos: UIButton, UISlider, …

- **Evento**: Objeto que describe el suceso ocurrido.

- **Acción** y **Target**: Es el método a ejecutar en el objeto objetivo.
  - Las acciones pueden implementarse usando más o menos parámetros:

```
func accion1()
func accion2(sender: AnyObject)
func accion3(sender: AnyObject, forEvent event: UIEvent)
```

© Santiago Pavón - UPM-DIT

Conectar Targets, Actions y Controles

- Usando **Interface Builder**:
  - Etiquetar en el fichero .swift la acción a ejecutar con **@IBAction**.
    - Conectar con el Control usando Ctrl-arrastrar
  - Inspector de conexiones:
    - Conectar las acciones a ejecutar con algún evento del control.
  - Conectar usando los popup de los controles.
  - etc…

Estos temas ya los practicamos en la **Demo: Hola Mundo**

- **Programáticamente**:

```
class UIControl : UIView {
    func addTarget(target: AnyObject?,
             action: Selector,
      forControlEvents controlEvents: UIControlEvents)

   func removeTarget(target: AnyObject?,
             action: Selector,
      forControlEvents controlEvents: UIControlEvents)
}
```

© Santiago Pavón - UPM-DIT

UIViewController

- En las aplicaciones normalmente se crea una pantalla distinta para cada tarea que tiene que hacer la aplicación:
  - Cada pantalla es una instancia de una subclase de **`UIViewController`**.
    - **`UIViewController`** es la clase base para crear pantallas nuevas.
      - Nos proporciona números métodos y propiedades útiles ya hechos.
      - Muchos de estos métodos los modificaremos para adaptarlos a nuestras necesidades usando herencia.
      - Con la herencia también añadiremos nuestra propia lógica.
    - Las clases VC que creemos para nuestras pantallas contendrán los datos, las vistas y la lógica necesarios para su funcionamiento.

- Y las aplicaciones suelen tener varias pantallas.

- Hay que tener en cuenta que existen terminales con distintos tamaños de pantalla.
  - Por ejemplo, un iPad tiene una pantalla más grande que la de un iPhone.
    - En la pantalla de un iPad podemos mostrar más información que en la de un iPhone.
    - El tipo de navegación entre pantallas debería ser diferente en un iPad y en un iPhone.
    - Rediseñar un GUI distinto para un iPhone y para un iPad.

© Santiago Pavón - UPM-DIT

- Para navegar entre las diferentes pantallas de una aplicación:
  - **Vistas modales**.
    - Una pantalla nueva tapa la pantalla actual.
  - **Barra de navegación** (Navigation Bar).
    - Navegar usando una pila (stack) de pantallas.
  - **Barra de Pestañas** (Tab Bar).
    - seleccionar pantallas independientes usando una especie de barra de pestañas.
  - **Split View Controller**.
    - Muestra dos VC, uno master y otro para detalles.
  - Otros:
    - Contenedores de View Controllers.
    - Aplicaciones basadas en páginas.
- No reinventar las formas de navegar:
  - Usar los patrones predefinidos.
  - Al usuario le será más familiar.
  - Ver las plantillas proporcionadas por Xcode.

© Santiago Pavón - UPM-DIT

Navigation Bar

Tab Bar

© Santiago Pavón - UPM-DIT

●●●●● movistar 13:12 5%

Buzones de entrada Editar

Buscar

Kinépolis 12/02/14
RoboCop llega a Kinépolis
Si no puede ver el mensaje correctamente, haga clic aquí KINÉP…

Movistar 11/02/14
¡sí, sin compromiso...!
movistar Si tiene problemas para leer este mensaje, consulte la versión en lí…

Mi Kinépolis 07/02/14
Tu selección personal de películas del…
Si no puede ver el mensaje correctamente, haga clic aquí Mi Kiné…

Google Cloud Platform 07/02/14
Updates to Cloud Platform Terms of S…
Dear Google App Engine customers, You are receiving this notification bec…

jsr 06/02/14
Re: Last chance! Get the new Web De…
Este mensaje está vacío.

Elena - eltenedor 06/02/14
¡En San Valentín el buen gusto es posi…
Los restaurantes más románticos de Madrid con promociones y menús Re…

Gabriel y Enrique 05/02/14

Actualizado justo ahora
2 mensajes sin enviar

Acción y ciencia ficción con RoboCop 2014

Estimado Santiago,

Protagonizada por Joel Kinamman, **"RoboCop"** vuelve a la gran pantalla de la mano de José Padilha el próximo **viernes 14 de febrero**.

En **"RoboCop"**, corre el año 2029 y el conglomerado multinacional OmniCorp está en el eje de la tecnología robótica.

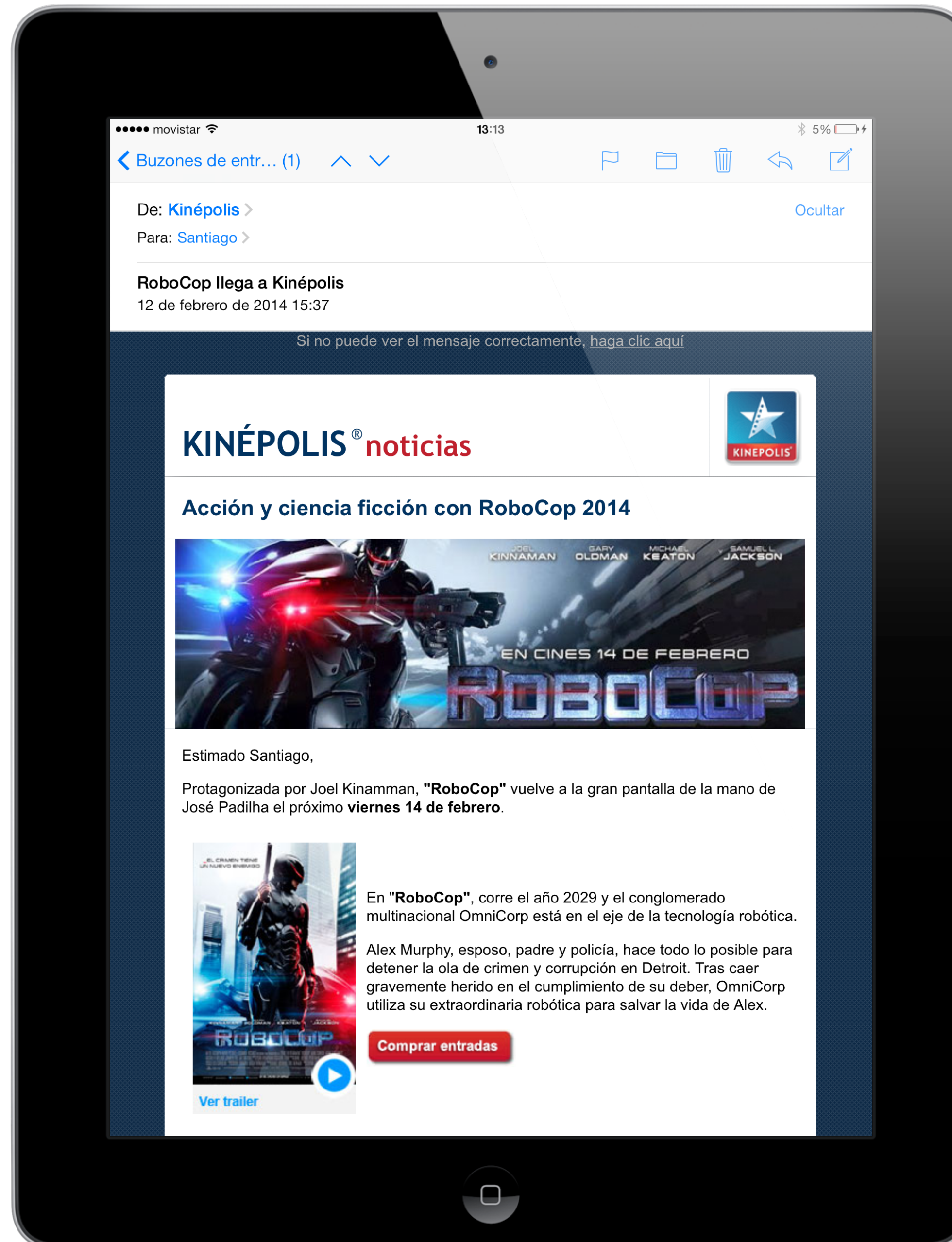
Alex Murphy, esposo, padre y policía, hace todo lo posible para detener la ola de crimen y corrupción en Detroit. Tras caer gravemente herido en el cumplimiento de su deber, OmniCorp utiliza su extraordinaria robótica para salvar la vida de Alex.

Comprar entradas

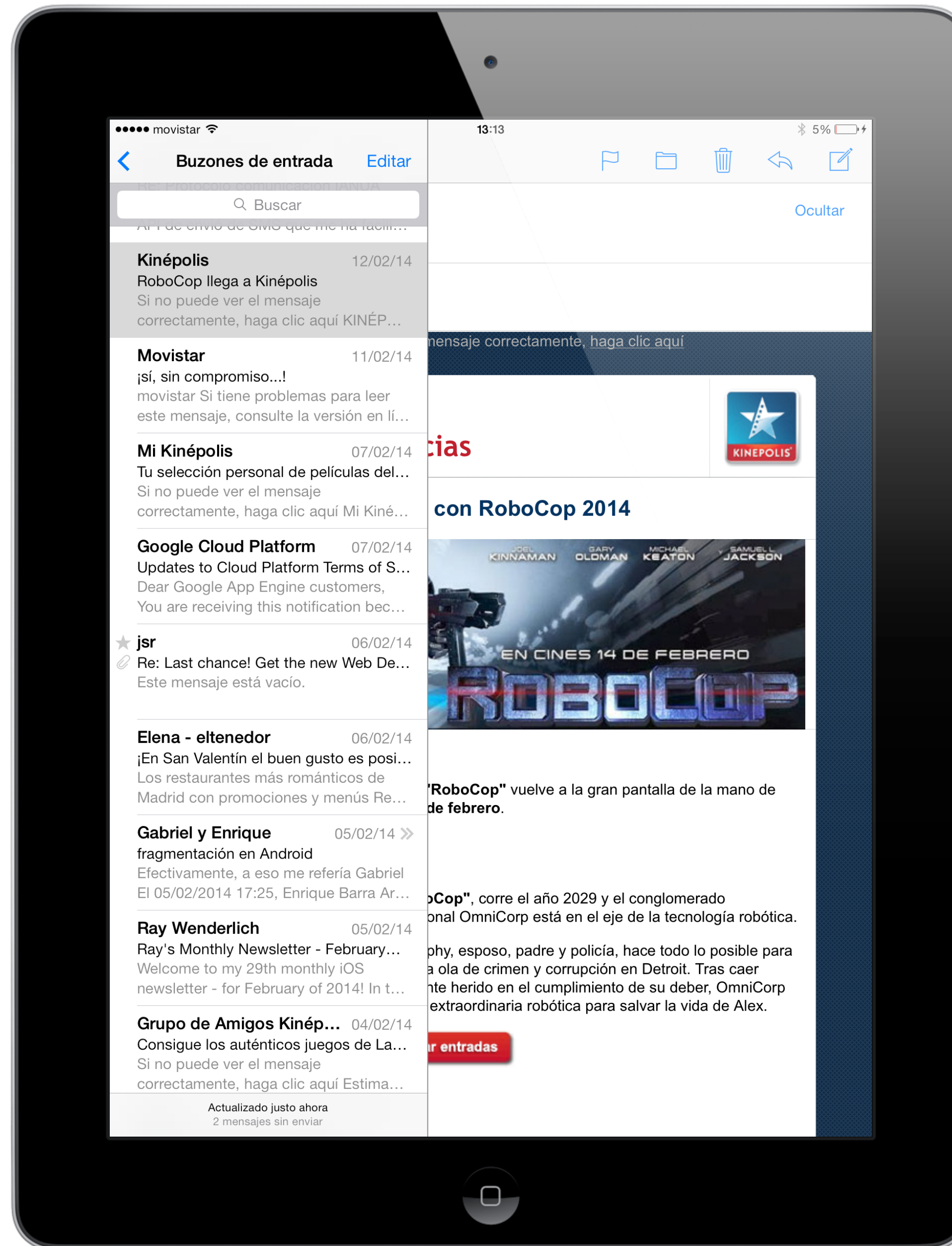
¡Participa en nuestro **concurso** con "RoboCop": Podrás ganar uno de los **3 Spybots** que regalamos, o una visita a **The Robot Museum**!

Click aquí para participar

© Santiago Pavón - UPM-DIT



© Santiago Pavón - UPM-DIT



© Santiago Pavón - UPM-DIT

Adaptabilidad

- En las versiones anteriores a iOS 8 y Xcode 6
  - Algunos elementos solo funcionaban en los iPads:
    - Split View Controller, Popover Controller, algunas opciones para las transiciones y las vistas modales, etc.
- En 2014 aparece:
  - iOS 8, Xcode 6, nuevos tamaños de terminales (iPhone 6 y iPhone 6 Plus), y el lenguaje: Swift.
  - Se introducen numerosos cambios:
    - PopoverController, SplitViewControllers, … funcionan en cualquier terminal.
    - La navegación entre pantallas se adapta según el tamaño de la pantalla: Por ejemplo, una app decide (sin programar nada especial) si usará un NavigationController o SplitViewController según el tamaño del terminal donde corra.
    - Más opciones de presentación: márgenes, comprimir barra de navegación, nuevas formas de presentar algunos controladores, etc…
    - . . .
- En 2016, con iOS 9 y Xcode 7, aparece multitasking que permite ver dos apps simultáneamente en pantalla. *(y más cosas)*
  - Es importante el diseño adaptativo.

© Santiago Pavón - UPM-DIT

Patrones

© Santiago Pavón - UPM-DIT

Creando una Aplicación

- Para hacer una aplicación:
  - Definimos varias clases para varios objetos que guardan los datos.
  - Creamos el GUI, que tiene numerosas etiquetas, botones, campos de texto, etc . . .
  - Nuestros controles llaman a varias acciones cuando el usuario los toca, actualizando los datos.
  - Según el estado de la aplicación determinados elementos los hacemos visibles, y otros los dejamos ocultos.
  - Cuando cambian nuestros datos actualizamos el GUI.
  - Y más cosas. Muchas más cosas que necesito para hacer mi aplicación.

- **Si este desarrollo se hace a lo loco, la aplicación resultante se parecerá a la representada en la siguiente transparencia.**

© Santiago Pavón - UPM-DIT

Este código no se puede mantener
Imposible probar cada parte
CAOS
No puedo reutilizar nada
Todo depende de todo

Patrón: Modelo-Vista-Controlador

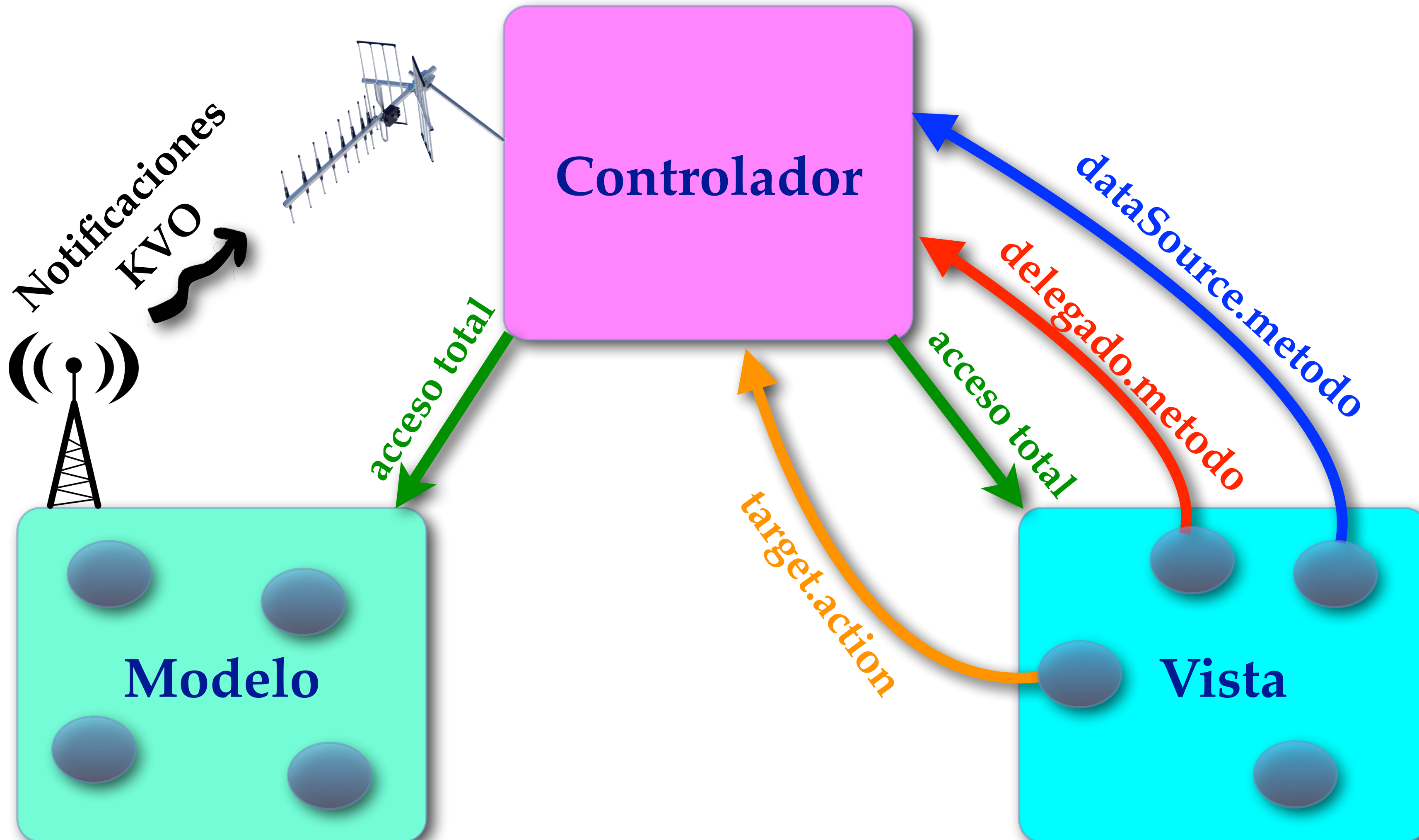
- Diseño más claro y fácil de mantener. Separar responsabilidades.
- Facilita reutilizar modelos y vistas.
  - El controlador difícil de reutilizar al estar muy ligado a la aplicación.
- Los modelos y vistas no deben verse entre sí.

- **Modelo**: Son los datos. Independientes de su forma de representación.
  - Para informar sobre cambios en los datos, el modelo difunde mediante KVO o notificaciones
    - El modelo no sabe nada de quien escucha.
    - Suele estar escuchando el controlador (y otros modelos). No las vistas.

- **Controlador**: Cómo se muestra el modelo al usuario. Contiene la lógica de la aplicación.
  - Intermediario ente M y V, adaptando lo que sea necesario. Tiene acceso total a M y a V.
    - Actualiza la vista si cambian los datos.
    - Actualiza el modelo según se manipule la vista.

- **Vista**: Objetos de presentación de uso general para uso del controlador.
  - Usados para representar los datos del modelo.
  - No poseen los datos.
    - Obtiene los datos preguntando a su data source. Debería ser el controlador, no el modelo.
  - Si el usuario manipula la vista, avisa al controlador usando:
    - Target-Action, delegación.

© Santiago Pavón - UPM-DIT

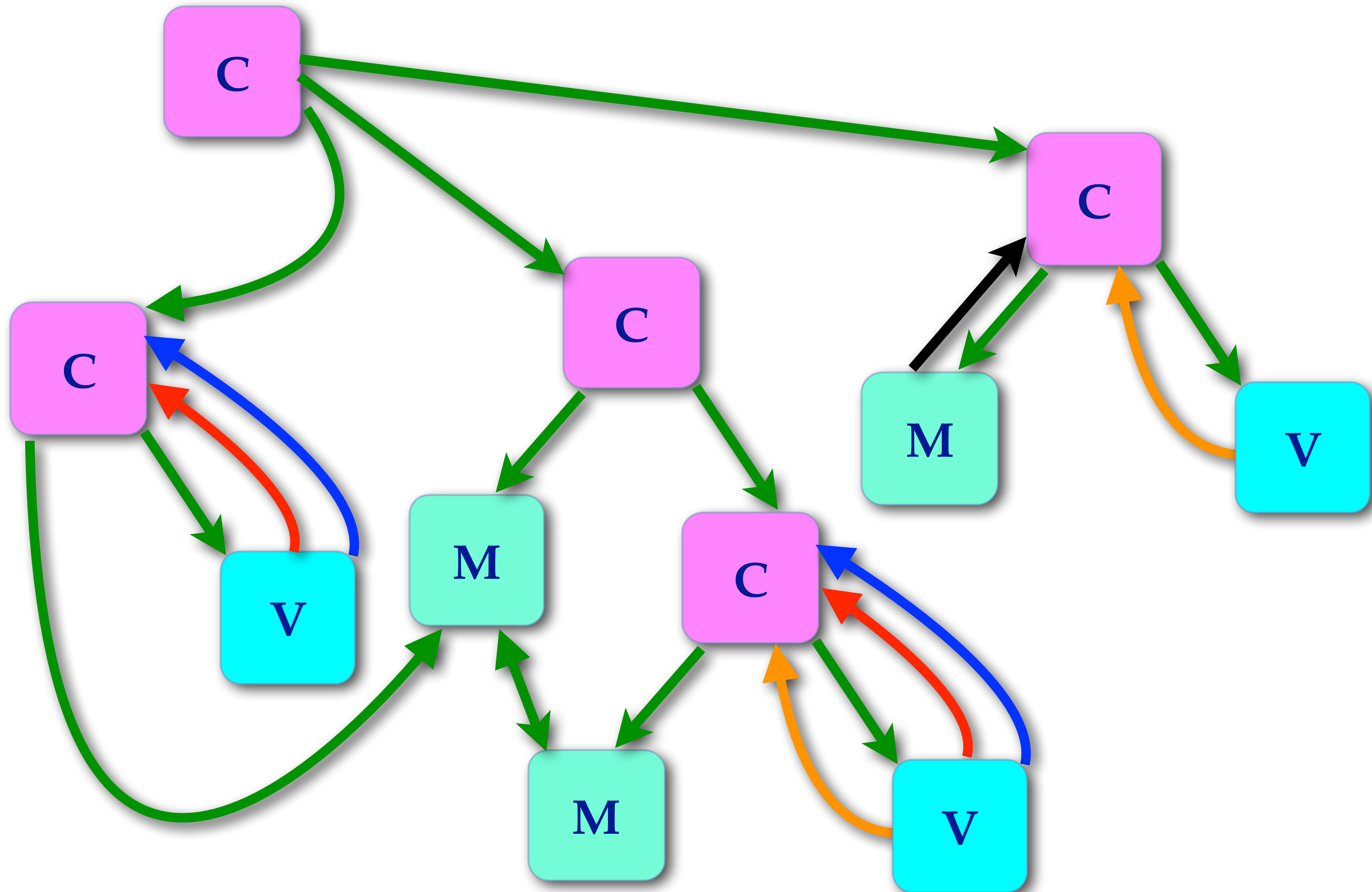


© Santiago Pavón - UPM-DIT

Colaboración de Varios MVC

- En una aplicación real existen varios MVC funcionando juntos.
  - La vista de un MVC la puede usar otro controlador.
  - Un modelo puede compartirse por varios controladores.
  - Puede existir acceso directo entre dos modelos.
  - Un controlador puede no tener modelo.

© Santiago Pavón - UPM-DIT



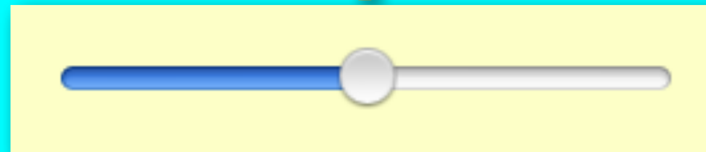
© Santiago Pavón - UPM-DIT

Target-Action

- La vista envía una acción a un objetivo.
  - no se sabe cual es la clase del objetivo.
  - para la vista el objetivo es de tipo **`AnyObject`**.

Controlador

```
@IBAction func sliderMoved(sender: UISlider) {

}
```

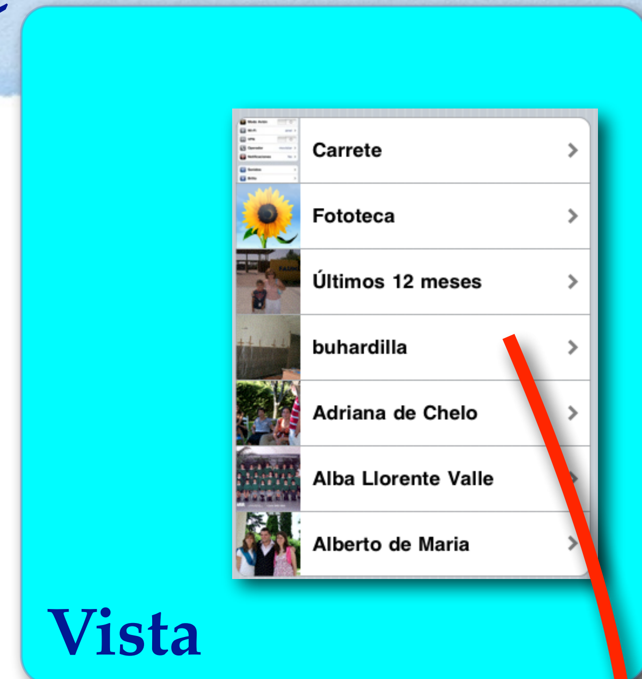


Vista

© Santiago Pavón - UPM-DIT

Delegate

- La vista necesita sincronizarse con su delegado.
  - Se define un protocolo con los mensajes que puede enviar.
    - Muchos mensajes son del tipo: **`did`***Algo*, **`will`***Algo* y **`should`***Algo*.
  - La vista sólo sabe que su delegado es un objeto conforme a ese protocolo.
- El delegado suele ser el controlador.



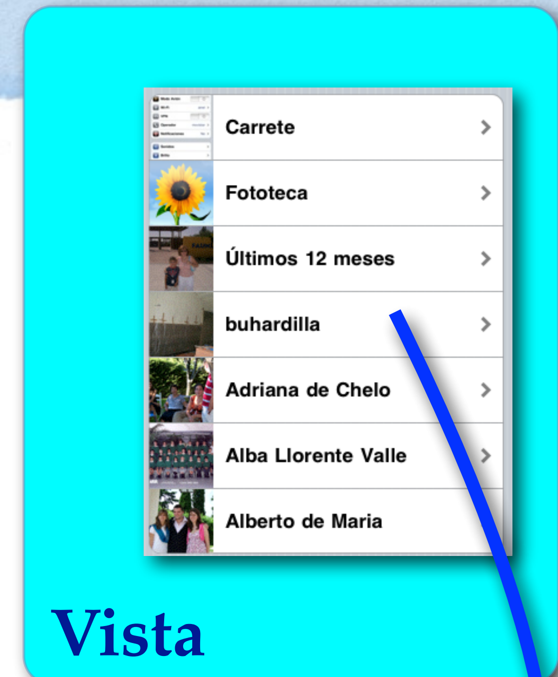
Controlador

```
func tableView(tableView: UITableView,
               accessoryButtonTappedForRowWithIndexPath indexPath: NSIndexPath)

func tableView(tableView: UITableView,
               willSelectRowAtIndexPath indexPath: NSIndexPath) -> NSIndexPath?

func tableView(tableView: UITableView,
               didSelectRowAtIndexPath indexPath: NSIndexPath)
```

© Santiago Pavón - UPM-DIT

Data Source

- Es el patrón de delegación, pero dedicado a la obtención de los datos.
- La vista no es la propietaria de los datos.
  - Los obtiene usando un protocolo que define los mensajes necesarios para obtener los datos.
  - La vista sólo sabe que los obtiene de un objeto conforme a ese protocolo.
- El controlador suele actúa como Data Source de la vista, adaptando los datos que coge del modelo.



Controlador

```
func tableView(tableView: UITableView,
               cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell

func numberOfSectionsInTableView(tableView: UITableView) -> Int

func tableView(tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int
```

© Santiago Pavón - UPM-DIT

KVO: Key-Value Observing

- El modelo no sabe nada del controlador.
- El controlador se registra como observador de una propiedad del modelo.
  - Cuando cambia el valor de la propiedad se notifica a los observadores.
- El observador implementa métodos para atender las notificaciones recibidas.

Notas: Swift no soporta KVO de forma nativa. Se usa KVO de NSObject.
El observador y el observado deben ser objetos NSObject.
Y si la propiedad a observar se ha declarado en Swift, debe marcarse como dinámica porque KVO modifica los métodos de acceso a ella.

© Santiago Pavón - UPM-DIT

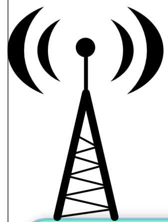
Controlador

```
// Variable global
private var miContexto = 0
```

```
datos.addObserver(self,
        forKeyPath: "counter",
           options: .New,
           context: &miContexto)
```

```
var datos = Modelo()
```

```
override func observeValueForKeyPath(keyPath: String!,
                                   ofObject object: AnyObject!,
                                change: [NSObject : AnyObject]!,
                               context: UnsafeMutablePointer<Void>) {
    if context == &miContexto {
        let d = object as Modelo
        println(d.counter)
        let i = change[NSKeyValueChangeNewKey]! as Int
        println(i)
    }
}
```



Modelo

```
class Modelo: NSObject {
    dynamic var counter = 1
}
```

© Santiago Pavón - UPM-DIT

Notificaciones

- Un centro de notificaciones se encarga de difundir información dentro de un programa.
  - Los objetos se registran en el centro de notificaciones indicando que notificaciones quieren recibir, y de quién las quieren recibir.
  - Otros objetos publican notificaciones en el centro de notificaciones para que éste avise a los objeto interesados en ellas.
- El centro de notificaciones por defecto se obtiene así:

```
NSNotificationCenter.defaultCenter()
```

© Santiago Pavón - UPM-DIT

- Para registrar en el centro de notificaciones a un objeto interesado en recibir notificaciones:

```
func addObserver(_ notificationObserver: AnyObject,
        // El objeto que recibirá la notificación
     selector notificationSelector: Selector,
        // Método a ejecutar cuando se recibe la notificación
     name notificationName: String?,
        // El nombre de las notificaciones que se desean recibir.
     object notificationSender: AnyObject?)
        // El objeto que envía la notificación, o nil para cualquier objeto.
```

- En el objeto observador se ejecuta el método selector especificado cuando recibe la notificación:

```
func metodoAEjecutar(notification: NSNotification) {
     notification.name     // El nombre de la notificación
     notification.object   // El emisor de la notificación
     notification.userInfo // Información sobre la notificación
}
```

- No olvidar eliminar del centro de notificaciones los objetos registrados cuando sea oportuno.
  - El centro de notificaciones no retiene a los objetos registrados en él.
    - Deben des-registrarse del centro de notificaciones antes de ser destruidos (dealloc).

```
func removeObserver(_ notificationObserver: AnyObject)
func removeObserver(_ notificationObserver: AnyObject,
               name notificationName: String?,
             object notificationSender: AnyObject ?)
```

- Los emisores crean y envían las notificaciones usando varios métodos:

```
func postNotificationName(_ notificationName: String,
                     object notificationSender: AnyObject?,
                   userInfo userInfo: [NSObject : AnyObject]?)
```

© Santiago Pavón - UPM-DIT

Controlador

```
var datos = Modelo()
```

```
let center = NSNotificationCenter.defaultCenter()

center.addObserver(self,
          selector: "receivedCounterNotification:",
              name: NotificationName,
            object: datos)
```

```
func receivedCounterNotification(notification: NSNotification) {
    if let m = notification.userInfo?["modelo"] as? Modelo {
        println("Recibida notificacion: \(m.counter)")
    }
}
```

```
let center = NSNotificationCenter.defaultCenter()

center.removeObserver(self)
```

```
let NotificationName = "CounterNotification"
```

```
var counter = 1
```

```
let center = NSNotificationCenter.defaultCenter()
let info = ["modelo": self]
center.postNotificationName(NotificationName,
                    object: self,
                  userInfo: info)
```

Modelo

© Santiago Pavón - UPM-DIT

© Santiago Pavón - UPM-DIT