POLITÉCNICA

ETSIT UPM

dit UPM

Desarrollo de Apps para iOS

Anatomía de una Aplicación

IWEB 2019-2020

Santiago Pavón

ver: 2019.11.03

Anatomía de una Aplicación

- Una aplicación está compuesta de:
  - El código generado por el programador (nuevos tipos, clases derivadas,...)
  - Frameworks usados: Foundation, UIKit, MapKit, . . .
  - Ficheros Storyboard.
  - Recursos: catálogo de activos (xcassets): imágenes, iconos.
  - Ficheros de configuración: info.plist
  - . . .
- El Código de la Aplicación se crea partiendo del código generado por las plantillas proporcionadas por Xcode.
  - Luego añadiremos nuestro código para implementar las funcionalidades de nuestra aplicación.
  - Editaremos con Interface Builder el fichero storyboard para crear parte del GUI.
  - Editaremos y configuraremos los ficheros de soporte generados por Xcode.
  - etc.

Ciclo de Vida de una Aplicación

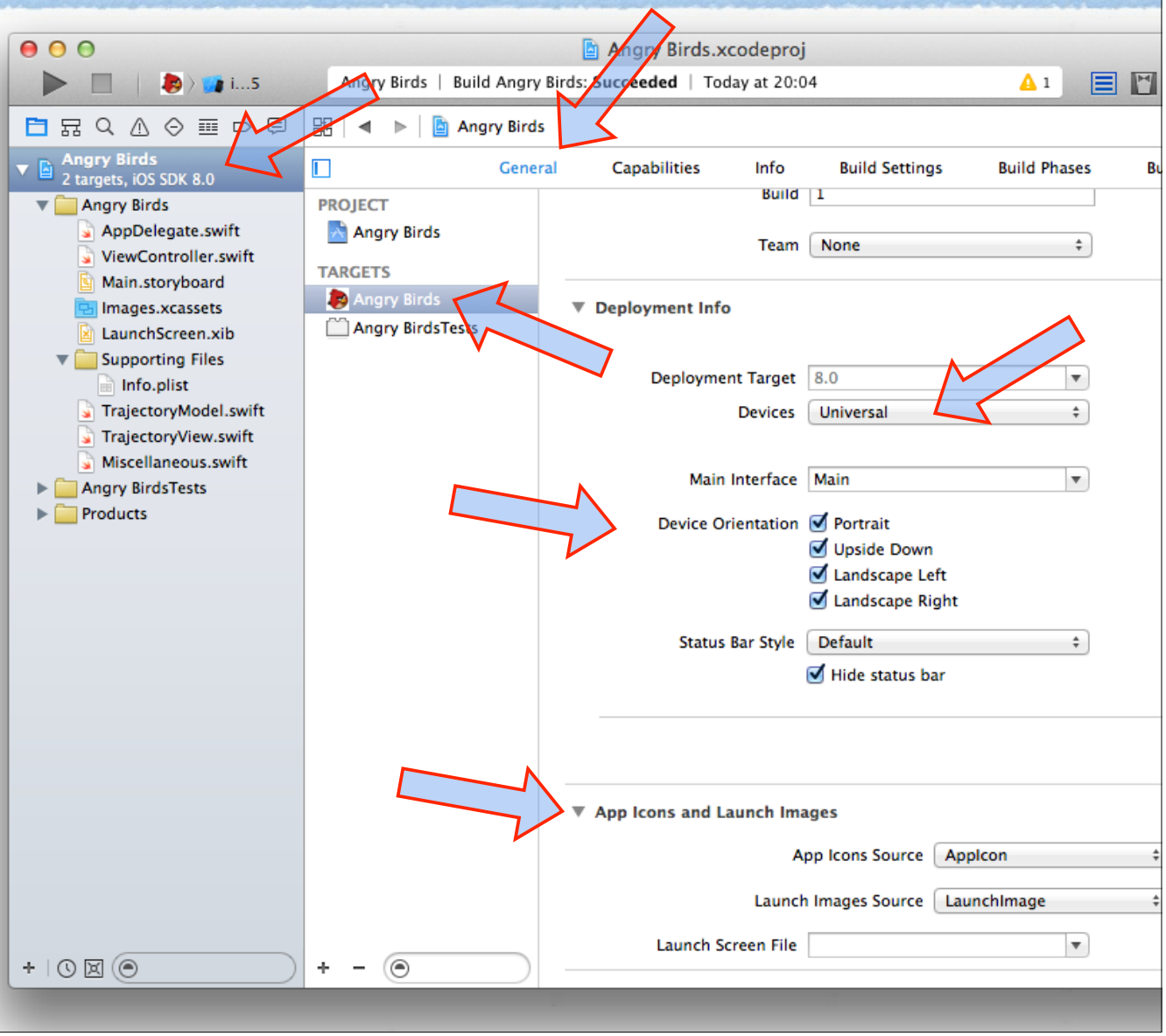
- **UIKit** crea la aplicación y controla su ciclo de vida.
  - Crea una instancia de UIApplication.
    - e instancias de otras clases.
  - Carga fichero storyboard principal.
  - Atiende los eventos.
    - Se ejecutan los manejadores (métodos) asociados a los eventos.
  - Gestiona la barra de estado.
  - Manejo de las interrupciones.
  - etc.

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`**.
- . . .

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.
  - . . .



UIApplication

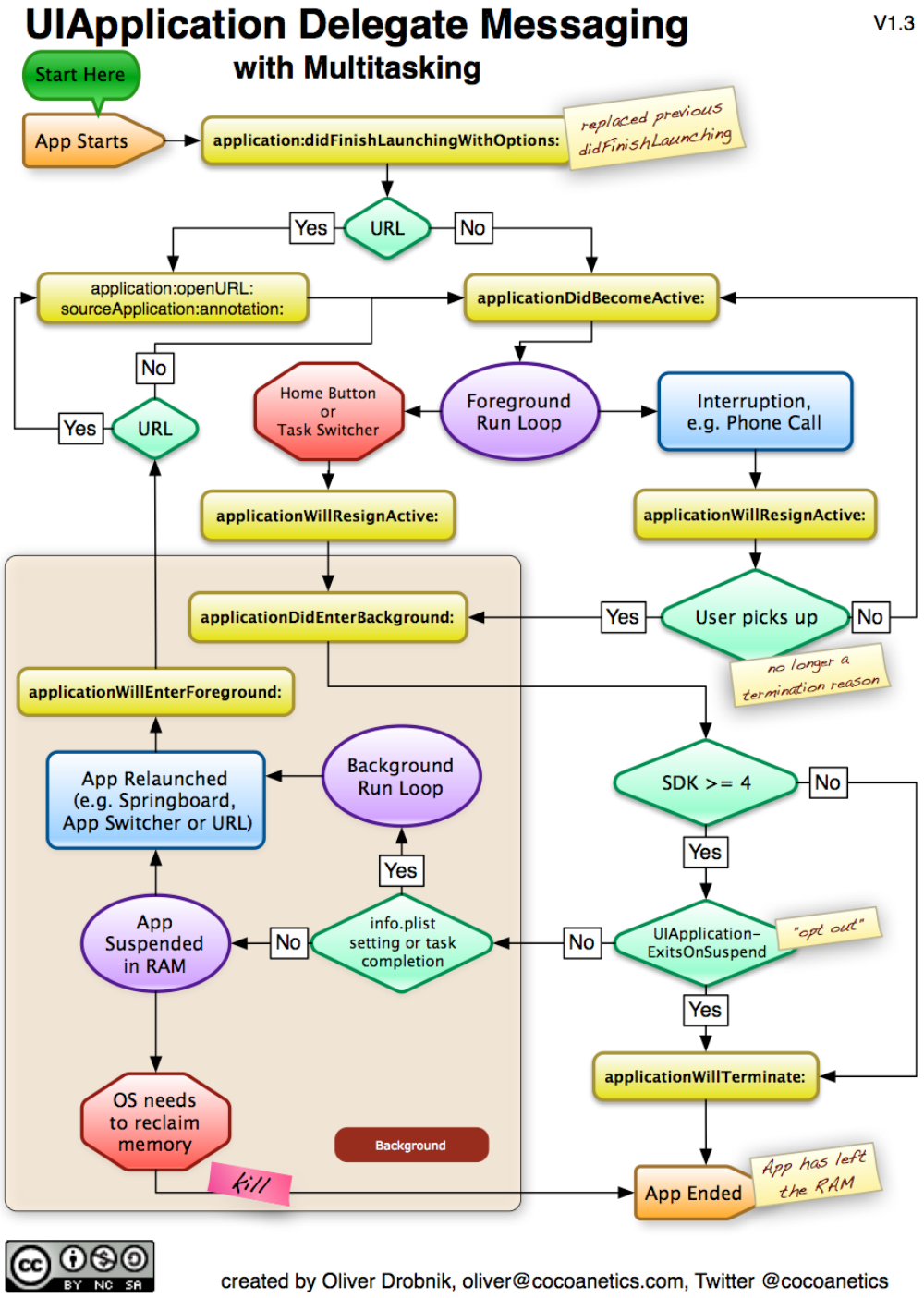
- Todas las aplicaciones tienen un único objeto **UIApplication** desde donde se controlan.
- Es un singleton al que se accede con:

```
UIApplication.shared
```

  - 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.

UIApplicationDelegate

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

  var window: UIWindow?

  func application(_ application: UIApplication,
          didFinishLaunchingWithOptions launchOptions:
                  [UIApplicationLaunchOptionsKey: Any]?) -> Bool {}

  func applicationWillResignActive(_ application: UIApplication) {}

  func applicationDidEnterBackground(_ application: UIApplication) {}

  func applicationWillEnterForeground(_ application: UIApplication) {}

  func applicationDidBecomeActive(_ application: UIApplication) {}

  func applicationWillTerminate(_ application: UIApplication) {}
}
```

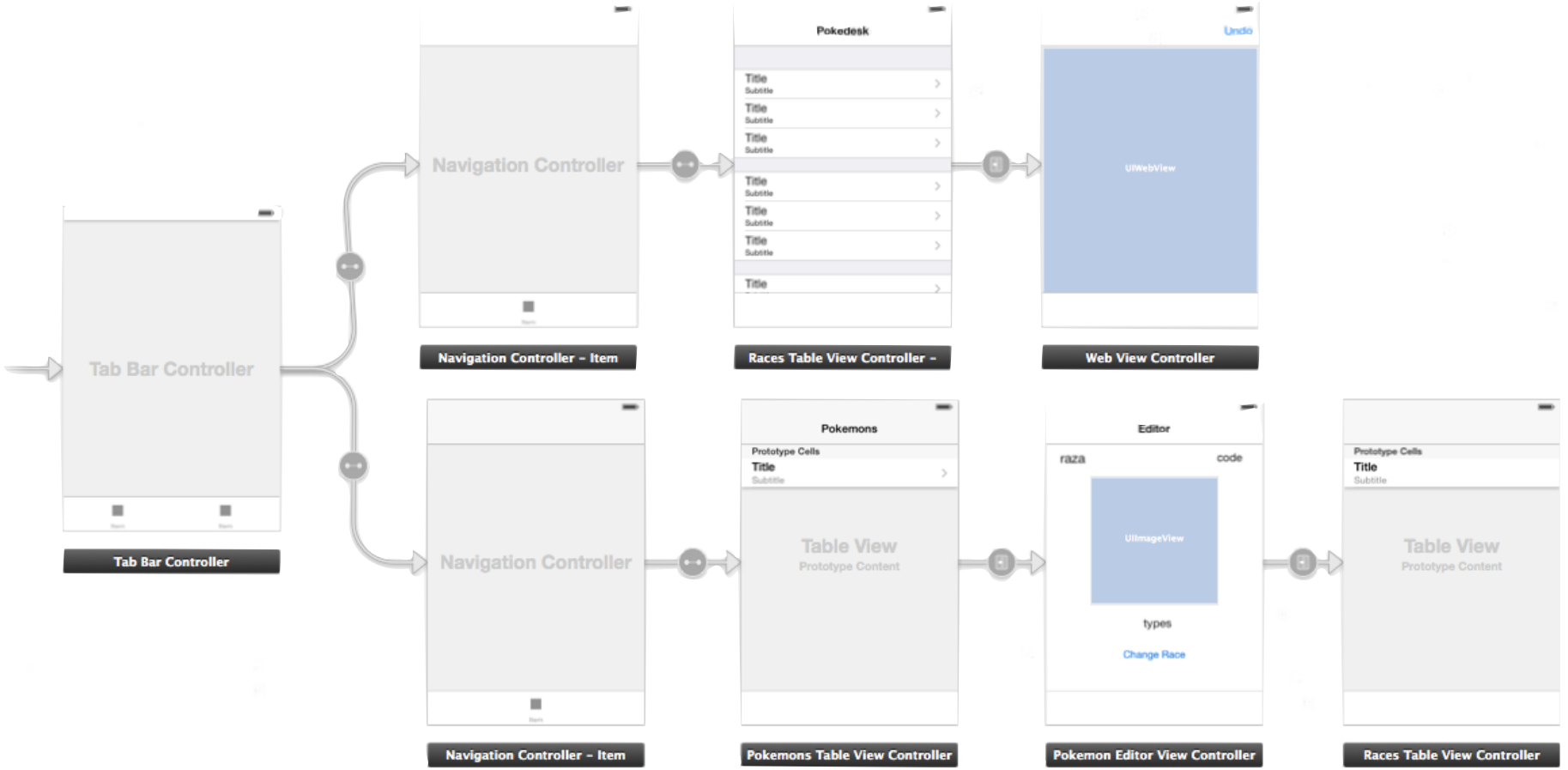


UISceneDelegate, UIWindowSceneDelegate

- Con Xcode 11 aparecen **UIWindowSceneDelegate** y **UISceneDelegate**
  - Gestionar tareas relacionadas con el ciclo de vida de las escenas.

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.



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.

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, …
```

Creación Manual de un GUI

- Un ejemplo: añadir en un `UIViewController`:


```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let r = CGRectMake(x:10, y:10, width:100, height:100)

        let l = UILabel(frame: r)

        l.text = "hola"
        l.transform = CGAffineTransform(rotationAngle: 0.5)

        view.addSubview(l)
    }
}
```

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: Any)
func accion3(_ sender: Any, forEvent event: UIEvent)
```

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**

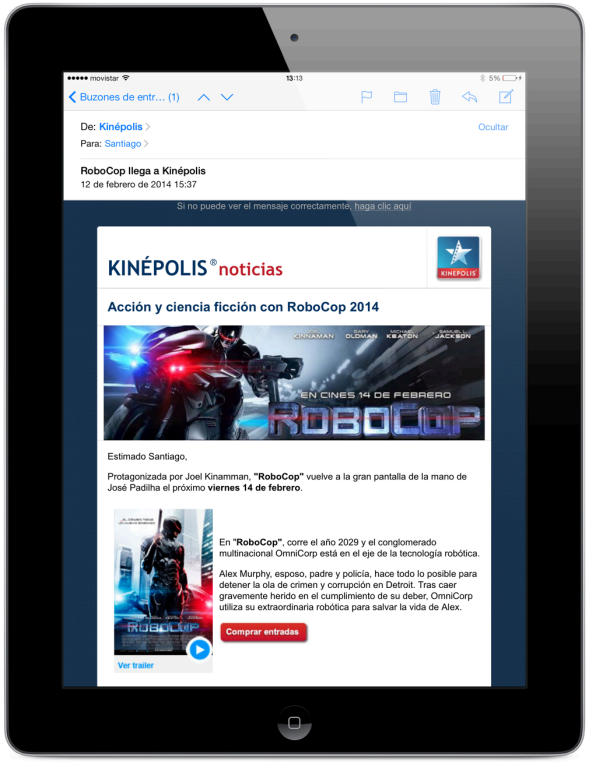
- **Programáticamente**:

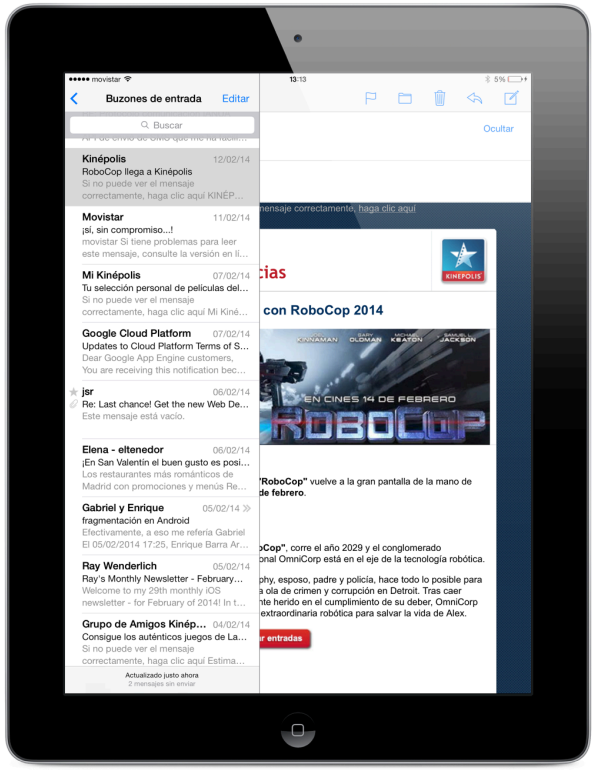
```
class UIControl : UIView {
    func addTarget(Any?,
                   action: Selector?,
                   for: UIControlEvents)

    func removeTarget(Any?,
                      action: Selector?,
                      for: UIControlEvents)
}
```

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 según el tamaño del terminal.
    - Diseñar el GUI para se adapte a los distintos tamaños de terminal.

- 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.





Patrones

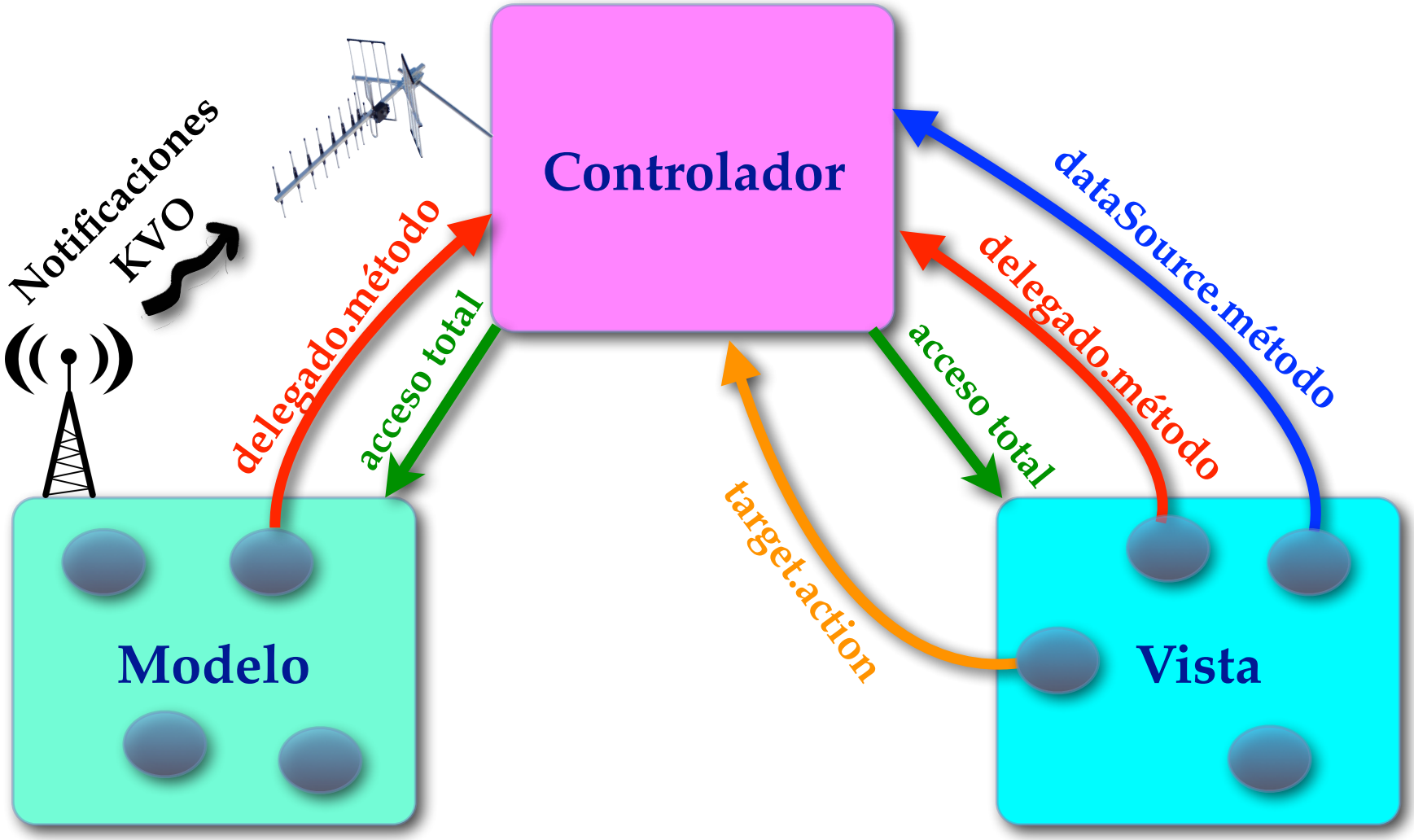
Patrón: Modelo-Vista-Controlador

- 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, notificaciones, delegación, callbacks, …
    - El modelo no sabe nada de quien escucha.
    - Suele estar escuchando el controlador (y otros modelos). No las vistas.

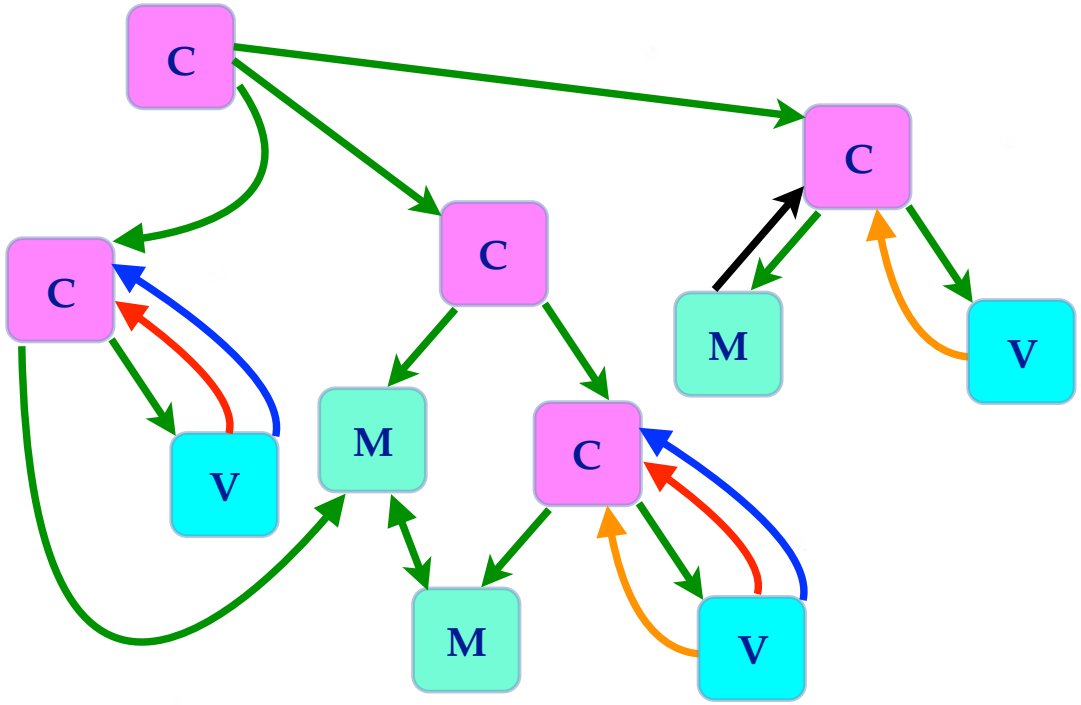
- **Controlador**: Cómo se muestra el modelo al usuario. Contiene la lógica de la aplicación.
  - Intermediario ente M y V, adaptando lo que sea necesario. Tiene acceso total a M y a V.
    - Actualiza la vista si cambian los datos.
    - Actualiza el modelo según se manipule la vista.

- **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.



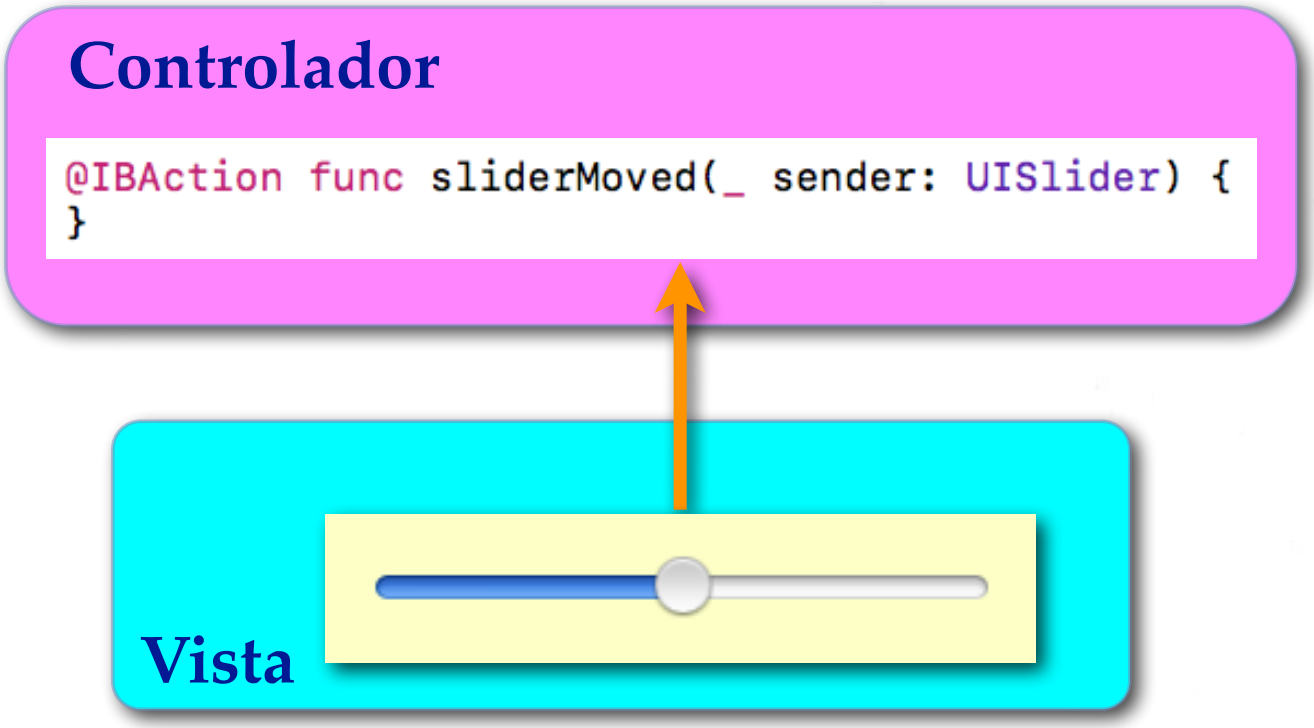
Colaboración de Varios MVC

- 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.



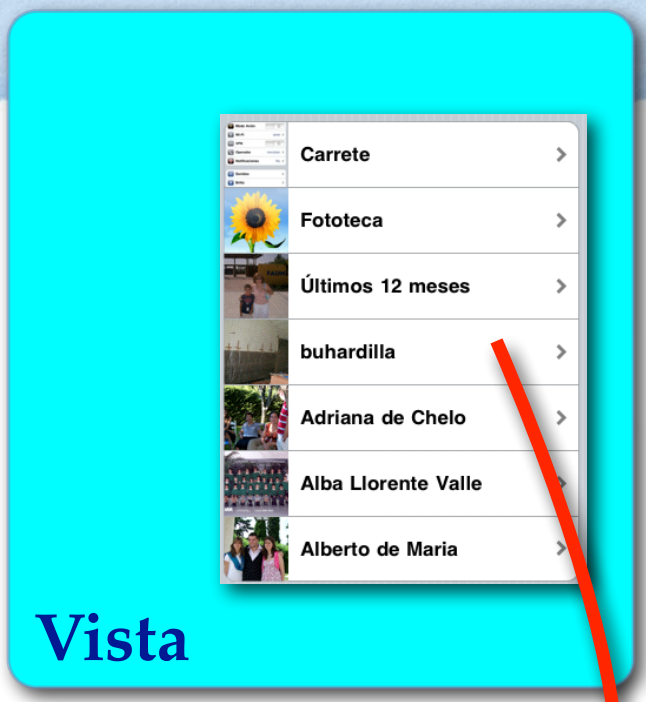
Target-Action

- 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`**.



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,
               accessoryButtonTappedForRowWith indexPath: IndexPath)

func tableView(_ tableView: UITableView,
               willSelectRowAt indexPath: IndexPath) -> IndexPath?

func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath)
```

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,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell

func numberOfSections(in tableView: UITableView) -> Int

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